MASTER EN SCIENCES INFORMATIQUES

Modélisation d'un système de messagerie sous la forme d'un "Domain specific language"

Hellebaut, Benoît

Award date:
2008

Awarding institution:
Universite de Namur

Table des matières

Résumé et mots-clés

Ce travail a pour objet de définir un *Domain Specific Language* visant à décrire une implémentation d'un middleware orienté message. Le middleware est créé à partir de composants élémentaires. Ces composants élémentaires sont le résultat d'une modélisation des patterns décrits dans le livre de Gregor Hohpe et Bobby Woolf, *Enterprise Integration Patterns*

Les composants qui implémentent le middleware sont chacun chargés d'une tâche élémentaire du middleware. On y retrouve des *Channels* chargés du transport et du stockage des messages, des *Routers* qui orientent les messages et encore des *Filter*s qui les valident. Enfin, on trouve les composants à l'interface du système qui prennent en charge le dialogue entre les applications et le middleware : les *Endpoints*.

Mots-clés : Intégration, middleware, message, Domain Specific Language, modélisation.

Remerciements

Je tiens à remercier Monsieur le Professeur Vincent Englebert pour l'attention apportée à ce travail et la grande patience qu'il a eue à mon égard. Je n'ai malheureusement pas pu toujours respecter le planning convenu en raison de quelques problèmes de santé. Je le remercie également pour le temps qu'il a pu trouver en cette fin de deuxième session d'examens malgré un emploi du temps fort chargé.

Je tiens également à remercier Mademoiselle Isabelle Linden pour tous les conseils donnés, sa patience et son soutien.

Enfin, merci à ma famille pour ses encouragements et son dévouement pendant toutes ces années et la rédaction de ce travail.

Chapitre 1

Introduction

1.1 Problématique

Pour des raisons historiques, une entreprise utilise plusieurs programmes informatiques comme par exemple une application de gestion des clients, un système d'enregistrement des services prestés, ... Ces différentes applications détiennent souvent des informations importantes voire stratégiques pour cette entreprise.

Des études de qualité montrent qu'une même donnée peut être différente selon le système d'informations : le nom d'un client n'est pas orthographié de la même façon ou son adresse n'est pas la même dans toutes les bases de données. Souvent également, la société souhaite avoir une image plus complète de ses clients afin de mieux cibler son offre de services, cette image étant le fruit de données réparties au sein de l'entreprise.

Ces deux raisons montrent le besoin de plus en plus criant pour l'entreprise d'intégrer ses applications et leur permettre d'échanger de l'information par exemple afin de signaler automatiquement le déménagement d'un client à toutes les applications qui utilisent cette donnée.

Cet échange n'est pas toujours évident pour des raisons de compatibilités de langage, Cobol ou Java, de plate-forme, Windows ou mainframe voire même de format de données comme la date écrite au format jour/mois/année dans certains systèmes ou au format année/mois/jour dans d'autres.

1.2 Middleware

Dans un premier temps, afin qu'elles puissent échanger de l'information, deux applications ont été modifiées et connectées. Ce mode d'échange dit

"point à point" (une application discute avec une autre application) a rapidement montré ses limites : une application A a développé un type d'échange avec une application B, puis un second type avec C, ... Le nombre d'interconnexions d'une application avec le reste de l'entreprise a ainsi rapidement donné l'image d'un plat de spaghettis, sans parler des modes de dialogue utilisés : échange de fichiers avec une application, accès à une base de données partagée par plusieurs systèmes d'informations.

Suite à ce problème de multiplication de connexions, une autre solution a été mise en place. Cette solution passe par l'utilisation d'un programme tiers, en anglais un "middleware". Cette solution consiste pour une application à transmettre l'information à échanger à cet intermédiaire qui la distribue ensuite aux applications intéressées. Le middleware peut aussi être utilisé par une application pour obtenir de l'information : l'application envoie sa demande d'information au middleware qui va transmettre la demande à l'application concernée qui répondra en utilisant de la même façon cet intermédiaire.

Pour permettre aux applications de dialoguer avec lui, le middleware est équipé de connecteurs. Un connecteur est un protocole qui décrit le mode de communication entre deux applications. Dans le cadre d'un middleware, il s'agit d'un protocole entre une application et le middleware : comment établir une connexion, transmettre ou demander de l'information, fermer la connexion.

Au sein d'un middleware, les informations transmises par une application sont encapsulées en une entité, un *Message*. Un message est généralement constitué de texte. Ce texte peut également être structuré voire être composé d'un flot de bits. Nous pouvons par exemple avoir le message *AddresseClient* qui contient l'adresse d'une personne sous la forme du texte libre *rue Grandgagnage, n°12 à 5000 Namur*.

Dès que l'application a fourni les informations au middleware, celui-ci prend en charge le routage du message, c'est-à-dire le choix de sa destination, ses éventuelles transformations et changements de format de données et de stockage avant sa livraison à l'application destinataire. Comme la transmission du message de composant en composant se fait en fonction de la disponibilité de chacun d'eux et de celle de l'application, on parle de transmission asynchrone : l'application envoie sa demande d'informations et continue son travail sans attendre la réponse. Elle recevra la réponse plus tard comme lors de l'échange de courrier postal. Un tel middleware qui transporte l'information, la route et la stocke avant livraison est appelé un *Middleware Orienté Message* ou encore *MOM*.

Il existe à l'heure actuelle, de nombreuses implémentations de middleware orienté messages mais toutes se révèlent coûteuses en ressources matérielles

et humaines. A la lecture du livre de Gregor Hohpe et Bobby Woolf, *Enterprise Integration Patterns*, nous avons pensé qu'il devrait être possible de proposer une description de ce type de solutions en posant qu'un MOM pouvait être constitué de composants élémentaires qui correspondent aux patterns proposés dans cet ouvrage. Chacun des ces composants réaliserait alors une tâche élémentaire telle que le transport d'un message d'un composant à un autre, les *Channel*s, le choix de la destination d'un message, les *Router*s ou encore la modification d'un message ou le filtrage des messages, rôle remplit par les *Filter*s.

1.3 Modélisation et *Domain Specific Language*

Sur base de ce livre et de notre exemple, nous proposerons un modèle de système. Un modèle selon les théories du [**MOF**] (un groupe chargé des spécifications de modélisation) est une représentation du monde réel. En revenant à notre exemple, notre modèle sera représenté par un schéma qui utilisera les pictogrammes proposés. On y retrouvera par exemple un *Channel* qui envoie le message reçu vers deux composants.

Cette image nous présente une solution d'intégration bien particulière à un problème. Comme il existe de nombreux problèmes d'intégration, nous pouvons ensuite généraliser les éléments précis pour en déduire des classes d'objets. Une classe d'objet permet de désigner un objet bien précis quand on applique des valeurs à ses options.

En généralisant les composants de notre schéma, nous produisons un *Meta Modèle* car ses éléments servent à décrire le modèle initial. Nous reviendrons plus en détail sur ces notions de modèle et de méta-modèle.

Pour permettre une mise en oeuvre et la maintenance d'un tel système, nous devrions proposer à son utilisateur un langage qui serve à décrire l'implémentation qui corresponde à la solution du problème d'intégration. Nous pourrions utiliser un langage de programmation générale, Java, .Net, ... comme celui utilisé pour l'implémentation des différents composants mais cela conduirait à une certaine forme de régression : au lieu de continuer à parler en termes de *Router*, *Channel*, nous reviendrons à parler de structure, d'algorithme.

Cette description serait plus aisée si le responsable de la mise en place du système d'intégration pouvait utiliser un langage dont les termes seraient les composants proposés. Ce type de langage, spécifique à un domaine bien particulier d'application, descriptif et qui concerne un nombre réduit d'objets, a été étudié notamment par van Deursen, Klint et Visser dans [**vanDeursenKlintVisser**]. Ils proposent la définition suivante pour un tel langage :

" A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. "

1.4 But de ce travail et méthodologie

Comme nous l'avons annoncé, le but de ce travail est de proposer un langage spécifique au domaine des middlewares orientés messages dont les éléments seront modélisés à partir des patterns proposés par [**Hohpe**].

Dans un premier temps, nous allons présenter un exemple de système de messagerie afin de familisariser le lecteur aux concepts d'un tel système. Cet exemple aura également pour but de présenter les pictogrammes proposés pour identifier rapidement chacun de ces éléments. L'exemple envisage le traitement d'un excès de vitesse de sa détection au règlement de l'amende pour excès de vitesse. Il présente plusieurs composants et leur rôle dans ce cadre.

Ensuite, nous allons passer en revue chacun des patterns décrits. Dans un premier chapitre, nous allons présenter les différentes catégories de composants. Puis nous passerons en revue en détail chacune d'entre elles. Pour chacun des patterns proposés, nous présenterons leur fonction. Nous y ajouterons une modélisation en langage UML et lorsque cela sera utile à la compréhension, des détails d'implémentation. En fin de catégorie, nous ajouterons lorsque cela est nécessaire quelques remarques concernant les éléments rencontrés. Ainsi, peu à peu, nous verrons apparaître l'architecture de l'ensemble du système, architecture qui sera la base de notre DSL.

Pour nous raccrocher aux types de solutions actuellement employées dans le monde informatique, nous en présenterons quelques-unes avec certains de leurs avantages et inconvénients. Nous reviendrons ensuite sur les notions de modèle, de méta-modèle et DSL.

Enfin, nous construirons le DSL et nous donnerons un exemple de système pour arriver à la conclusion de ce travail.

Chapitre 2

Scenario

2.1 Présentation de l'exemple

Nous avons choisi comme exemple d'intégration pour ce travail, l'automatisation du traitement d'un excès de vitesse, de sa détection à la réception du paiement de l'amende.

Cet exemple met en oeuvre plusieurs applications à intégrer : la collecte des infractions, l'identification du véhicule et le dialogue entre le propriétaire du véhicule et le Tribunal de Police chargé de juger de telles infractions.

Il est important de noter que cette description présente une situation vraisemblable mais n'est évidemment pas conforme à la réalité. Elle est destinée à introduire les différents composants du livre de [**Hohpe**] qui servent de base à la réalisation de cette étude.

2.2 Description générale du cas

Sur base de la vie courante, nous pouvons dire qu'un excès de vitesse est constaté par un radar automatique ou par un agent de police.

Suite à la constatation de cette infraction, le propriétaire du véhicule est identifié grâce au numéro d'immatriculation de son véhicule. Un procès-verbal, en abrégé P.V., est alors envoyé à ce dernier. Le P.V. contient les données de l'infraction c'est-à-dire le lieu et la date de l'excès de vitesse, la vitesse constatée et la vitesse autorisée à ce moment. Ce document permet au possesseur du véhicule de prendre connaissance des faits reprochés.

Nous poserons (hypothèse de simplification) dans cet exemple que le possesseur du véhicule est le contrevenant ; le fait que le possesseur du véhicule ne soit pas le conducteur au moment des faits pourra faire l'objet d'une

extension de l'exemple.

L'infraction est transmise au Tribunal de Police afin d'être jugée.

Après jugement et lorsqu'une amende est requise, les données relatives au paiement (montant de l'amende, personne qui doit régler celle-ci, ...) sont envoyées à la Poste qui est chargée d'obtenir le paiement de l'amende.

2.3 Cas d'utilisation

Le traitement d'un excès de vitesse peut être découpé en cas d'utilisation selon la figure 2.1.

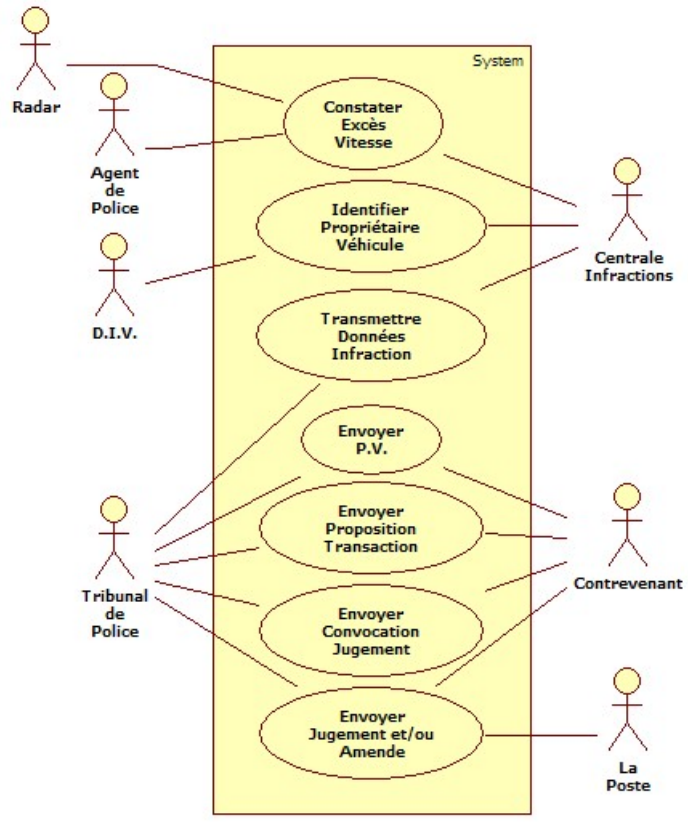

FIG. 2.1 – Cas d'utilisation : Excès de vitesse

Chapitre 3

Détail des cas d'utilisation et composants

3.1 Constater un excès de vitesse

Lors du constat d'un excès de vitesse, le radar ou l'Agent de Police, via une application d'encodage, envoie un message vers la Centrale des infractions, chargée de la réception de tous les constats d'excès de vitesse. Chaque message contient le numéro d'immatriculation du véhicule en infraction, la date et le lieu de l'infraction, la vitesse constatée et la vitesse permise à cet endroit.

Nous supposons dans cet exercice que le radar et la Centrale des infractions sont des applications indépendantes de notre système, localisées à des endroits clairement distincts.

Le système que nous voulons modéliser est chargé d'acheminer les messages entre ces deux applications. Le transfert des messages est confié selon [**Hohpe**] à un premier composant du système : le *Message Channel* dont le rôle est d'acheminer un message d'une application à une autre.

Le point d'entrée dans le système à utiliser par le radar, extérieur à notre système, est appelé un *Endpoint* ; il sert d'interface entre une application extérieure et le système que nous sommes en train de construire.

Ce cas d'utilisation nous permet donc de poser que :
- les applications (radar et Centrale des Infractions) sont des éléments extérieurs à notre système ;
- les composants (*Message Channel*) sont les éléments de notre système ;
- certains composants (*Endpoint*),à la frontière de notre système, permettent le dialogue entre les applications extérieures et notre système ;
- les applications dialoguent via le système de messagerie en échangeant

des messages.

3.2 Identifier le propriétaire d'un véhicule

La Centrale des Infractions est chargée dès la réception d'un constat d'excès de vitesse, de compléter les informations relatives à l'excès de vitesse avec les données du propriétaire du véhicule. Pour ce faire, elle interroge la D.I.V. Celle-ci retourne sur base du numéro de plaque les informations relatives au propriétaire du véhicule.

La requête à la D.I.V. et la réponse sont envoyées chacune de façon asynchrone et via des *Message Channel*s différents. Ainsi la Centrale peut continuer son travail (notamment la réception d'autres constats) sans devoir attendre la réponse de la D.I.V. qui peut nécessiter un certain temps. Cela peut être le cas par exemple d'une application peu informatisée et qui nécessite une action humaine. Le traitement de la réponse sera fait par la Centrale quand elle aura reçu la réponse à une requête.

Comme la requête et la réponse sont envoyées de manière asynchrone, il est nécessaire d'utiliser un mécanisme pour permettre de faire la corrélation entre les deux messages. Nous utilisons pour faire la corrélation entre les deux messages deux composants :

- un *Content Enricher* dont le rôle est d'ajouter à la requête un identifiant unique ; identifiant qui sera placé dans la réponse par la D.I.V. ;
- un *Correlation Identifier* qui utilise l'identifiant pour faire le lien entre la requête et la réponse, messages asynchrones.

3.3 Transmettre les données de l'infraction

La transmission des données de l'infraction complétées de l'identification du propriétaire du véhicule à un Tribunal de Police peut ensuite se faire de deux manières différentes selon les rôles attribués aux applications :

- soit la Centrale des Infractions complète les informations relatives au propriétaire du véhicule à l'aide de la réponse de la D.I.V. et envoie un message au Tribunal contenant les informations de l'excès de vitesse et les informations du propriétaire du véhicule sous la forme d'un seul message ;
- soit la Centrale envoie les informations relatives à l'excès de vitesse d'une part à la D.I.V et d'autre part à un nouveau composant qui sera chargé de faire l'agrégation du message *ExcèsVitesse* et du message *IdentificationPropriétaire* qu'il recevra ultérieurement de la D.I.V. Ce composant enverra alors le message résultat vers le Tribunal de Police.

Nous avons déjà décrit la première possibilité de réaliser ce traitement dans le cas d'utilisation précédent. Par contre, la seconde manière de procéder nous permet :

- d'introduire un nouveau type de composant : le *Publish-Subscribe Channel* qui reçoit un message et le diffuse à plusieurs destinataires ;
- de mettre en évidence une caractéristique du *Message Channel* : le découplage entre l'émission du message et sa transmission. En modifiant la destination du message envoyé par la D.I.V. au niveau du *Channel*, nous ne changeons rien au traitement de la D.I.V. ;
- nous introduisons également un nouveau type de composant : l'*Aggregator* qui reçoit des messages de plusieurs origines diffférentes et les aggrège pour n'en faire plus qu'un.

Enfin, le *Content Filter* est utilisé pour filtrer le contenu du message émis par la Centrale pour ne garder que le numéro de plaque et l'identifiant de la demande. Le message résultant étant plus léger et plus significatif pour l'application destinataire.

3.4 Envoyer le procès-verbal

Le Tribunal envoie au propriétaire du véhicule un message reprenant les informations reçues de la Centrale des Infractions (ou de l'*Aggregateur*). Comme le Tribunal ne peut connaître les "adresses" de tous les propriétaires de véhicule, il délègue cette tâche au *Content-based Router* qui envoie le message vers le propriétaire sur base des informations contenues dans la partie "Adresse" du message.

3.5 Proposer une transaction

La proposition d'une transaction consiste en l'envoi d'un message reprenant les données de l'infraction et un montant à payer, le tout accompagné d'un talon pour signifier l'approbation ou le refus de la transaction.

L'envoi de cette proposition se fait suivant le même mécanisme que celui utilisé pour l'envoi du procès-verbal. Le message de proposition de transaction est envoyé vers deux destinations simultanément :

- le propriétaire du véhicule et
- un composant qui enverra un message après l'écoulement d'un certain temps.

En répondant, par acceptation ou par refus, le propriétaire du véhicule en infraction envoie à son tour un message (voir le talon de réponse). Ce message est transmis via un *Message Channel* vers le Tribunal. Au-delà d'un certain

délai (voir à cet effet la législation en cours), le second destinataire de la proposition, envoie lui aussi un message vers le Tribunal ; ce message sera le signal que le temps écoulé pour répondre à la proposition de transaction aura été dépassé.

Le Tribunal pourrait alors recevoir deux messages pour une seule infraction. Afin d'éviter ce désagrément et un travail inutile au sein du Tribunal, nous installons dans la chaîne de retour un nouveau composant proposé par [**Hohpe**] : le *Message Filter* dont le rôle est de supprimer le message excédentaire.

3.6 Envoyer la convocation pour le jugement

L'envoi de la convocation lorsque l'infraction ne peut faire l'objet d'une transaction peut se faire de la même façon que l'envoi de la proposition d'une transaction.

3.7 Envoyer le jugement et l'amende

L'envoi d'un jugement se fait comme nous avons procédé pour l'envoi du procès-verbal : le Tribunal envoie un message reprenant les termes du jugement (infractions, rappel de la convocation, éventuellement le refus de la transaction proposée et le texte du jugement proprement dit). Le message produit est dirigé vers le propriétaire du véhicule.

Toutefois, nous pourrions également utiliser pour ce cas d'utilisation un *Publish-Subscribe Channel* et un *Content-Filter* car le Tribunal doit également envoyer un message à la Poste afin que celle-ci se charge de la récupération du montant de l'amende dont devrait normalement s'acquitter le contrevenant.

3.8 Schéma d'implémentation

En utilisant les pictogrammes des éléments introduits, nous pouvons maintenant présenter à la figure 3.1 une vue globale des applications concernées et des messages échangés.

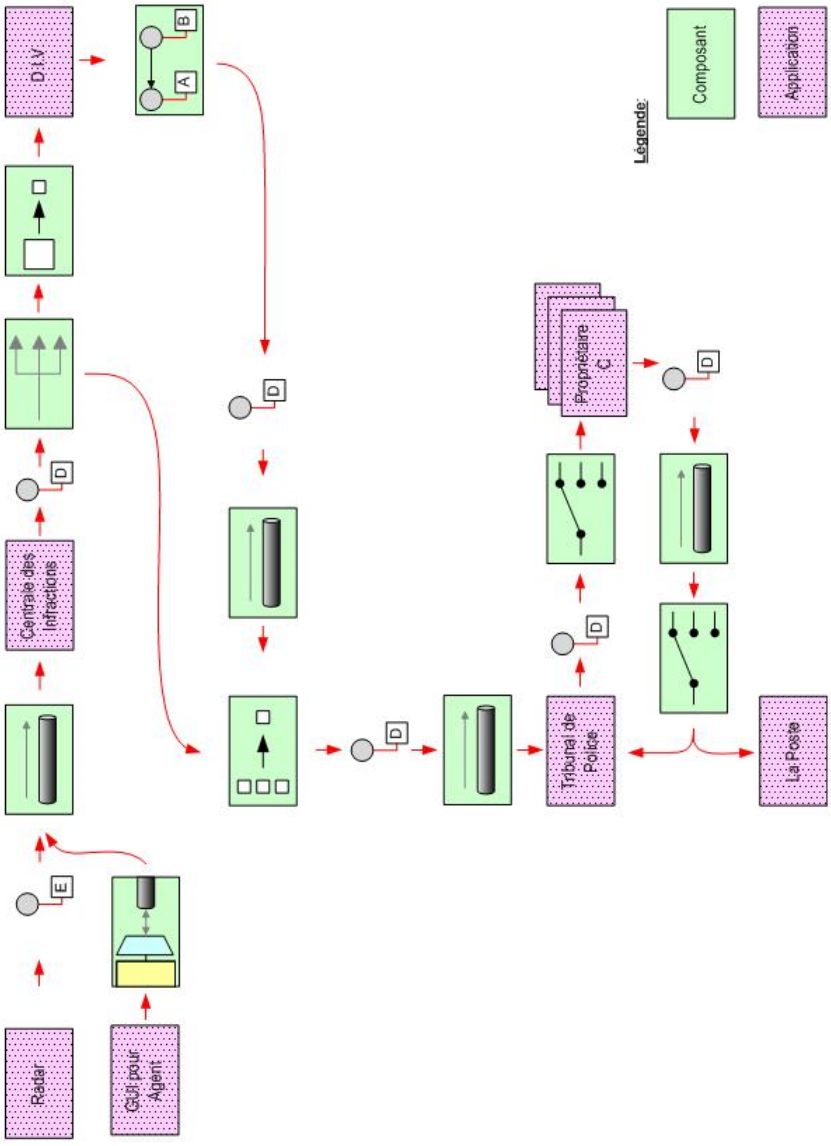

FIG. 3.1 – Applications, composants et messages

Chapitre 4

Aperçu des éléments du DSL

4.1 Introduction

En nous basant sur notre exemple, nous pouvons poser que le système est composé de composants.

Un composant peut :
- produire un message à partir de l'information mise à disposition par une application ;
- émettre un message vers un autre composant du système ;
- recevoir un message et le traiter et enfin
- utiliser un message pour transmettre de l'information à une application.

Nous posons que tous les composants du système ont comme base la classe abstraite *Component*. Cette classe possède comme attribut un nom(*name)*. Ce nom doit être unique au sein du système. Cette contrainte peut s'écrire en [**OCL**], un langage standardisé d'expression des contraintes, sous la forme :

```
context Component
inv: Component.allInstances()->
forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

De cette façon, quand nous parlerons dans cette étude de "composant", nous saurons qu'il s'agit d'une spécialisation de cette classe.

Avant de nous pencher sur les différentes catégories de composants proposés c'est-à-dire :
- le *Message Channel* permet de connecter deux applications entre elles ;
- le *Router* sélectionne un destinataire à chaque message reçu par application de règles ou de filtres ;

– le *Translator* modifie le message en le complétant ou en en modifiant le format afin de le rendre compréhensible par le destinataire ;
– le *EndPoint* agit à l'interface du système. Il permet à un producteur de messages d'introduire un message dans le système ou de le recevoir à travers le même système.

nous devons nous attarder sur le message lui-même qui, sans faire partie du DSL, doit être présenté car il est manipulé par les différents composants du système.

4.2 Message

4.2.1 Présentation

Un message est représenté par le pictogramme de la figure 4.1. Il est l'unité d'information échangée entre deux composants du système de messagerie que nous souhaitons modéliser.

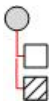

FIG. 4.1 – Pictogramme : message

Chaque message a un type. Trois types de messages sont prédéfinis :
– le *Command Message* ;
– le *Document Message* et
– l'*EventMessage*.

Chaque type est une indication du type d'usage du message.

Outre un type, [**Hohpe**] définit le message comme composé des éléments suivants :
– un *header* :
cet élément est destiné à contenir des informations décrivant le message ; les informations qu'il contient peuvent notamment être utilisées pour la transmission du message en contenant par exemple une adresse ;
– un *body* :
qui contient les informations proprement dites du message. Il s'agit en général de texte.

4.2.2 Modélisation

Le type du message est modélisé par la classe *TypeMessage*. Comme pour les noms de composants, nous pouvons exprimer que chaque type de message

possède un nom unique grâce à la contrainte OCL suivante :

```
context TypeMessage
inv: TypeMessage.allInstances()->
forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

L'en-tête est modélisé comme une liste d'éléments que nous appelons *Header* par extension. L'en-tête du message est également modélisé la classe *Header*. Afin de distinguer les éléments constitutifs de l'en-tête, nous allons poser que chacun d'entre eux possède un nom et une valeur. Un message peut contenir plusieurs éléments *Header* avec le même nom : lorsqu'on souhaite enregistrer le passage d'un message dans un composant en utilisant un *Header*, on pourra avoir un *Header* avec les valeurs (history, router) et un second *Header* avec les valeurs (history, channel). A priori, il n'existe pas de lien entre le type du message et les éléments qui composent son en-tête. Ces éléments constituent des informations pour le traitement par les composants du système.

Le message est enfin modélisé par la classe *Message*. Le message doit être sérialisable afin de pouvoir être transporté par les éléments du système. Afin d'identifier chaque message au sein du système, nous ajoutons au message un identifiant unique.

Nous pouvons finalement représenter le message par le diagramme de la figure 4.2.

FIG. 4.2 – Diagramme de classes : Message typé

4.2.3 Exemple

Dans l'exemple introductif, nous avons montré que le message peut contenir les coordonnées du propriétaire de cette voiture (Dupont, Charles, rue de la Gare, 23, Houtsiploux). Nous avons donc :

- type de message : "Document" ;
- contenu du message : "Dupont, Charles, rue de la Gare, 23, Houtsiploux" ;
- pas d'en-tête.

4.3 Message Channel

4.3.1 Présentation

Un *Message Channel*, représenté par le pictogramme de la figure 4.3, est chargé de transmettre un message d'un composant à l'autre de façon asynchrone. Il introduit un découplage entre le composant émetteur et le composant récepteur : chacun d'entre eux connaît uniquement son correspondant direct. Ceci permet de changer le destinataire du message sans devoir modifier l'émetteur du message.

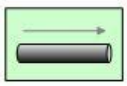

FIG. 4.3 – Pictogramme : Message Channel

Un *Message Channel* est notamment caractérisé par le nombre de destinataires à qui il envoie un message reçu, le type de données qu'il peut acheminer et la persistance des messages en cas d'arrêt du système.

4.3.2 Modélisation

A ce niveau du travail, le *Channel* est modélisé par une classe abstraite. Il a un seul point d'entrée. Les destinations des messages reçus sont configurées pour chaque *Channel*.

Les messages sont stockés dans le composant avant d'être traités par le composant. Ce mécanisme est utilisé pour assurer le traitement asynchrone des messages. La quantité maximale de messages dans le tampon d'entrée est enregistrée dans l'attribut *maxIncomingMessages*. La valeur de cet attribut sera configurée au niveau des paramètres du système.

Les messages en attente de livraison par le composant vers le destinataire sont également stockés en interne. La quantité maximale de messages est également paramétrable. Elle est enregistrée dans l'attribut *MaxOutgoingMessages*.

4.3.3 Exemple

Un exemple de *Message Channel* est présenté dans l'exemple introductif : le composant qui achemine les messages provenant des radars vers la Centrale des Infractions est un composant de ce type.

4.4 Message Router

4.4.1 Présentation

Le *Message Router*, représenté par le pictogramme de la figure 4.4, est le résultat de la combinaison d'un *Message Channel* et de filtres : le *Message Router* reçoit un message et le renvoie vers un destinataire choisi parmi une liste.

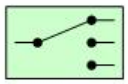

FIG. 4.4 – Pictogramme : Routeur

Le critère de choix du destinataire dépend du type de *Message Router*. Il se fait sur base :

- d'un état du routeur (pour une distribution de charge par exemple) ;
- du contenu du message ou encore
- du type de message.

4.4.2 Modélisation

Le *Router* est modélisé par une classe abstraite qui est elle-même une spécialisation du composant de base. Il possède un point d'entrée des messages et autant de points de sortie qu'il possède de destinataires. Ces destinataires sont fixés dans la configuration de ce composant.

Comme le *Channel*, le *Router* possède deux attributs facultatifs qui fixent la quantité maximale de messages en attente de traitement et de livraison.

4.4.3 Exemple

Dans l'exemple introductif, nous avons rencontré un *Router* pour choisir le destinataire du procès-verbal : ce router envoie un message vers le propriétaire du véhicule en infraction sur base de l'adresse fournie par la D.I.V.

4.5 Message Translator

4.5.1 Présentation

Le *Message Translator* est responsable de la transformation d'un message : il reçoit un message et en envoie un nouveau créé sur base du message reçu. La transformation se fait sur base de critères spécifiques à son type.

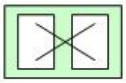

FIG. 4.5 – Pictogramme : Message Translator

Le *Translator* agit comme un interprète qui transforme les messages reçus de façon à les rendre compréhensibles par le destinataire. Lorsqu'une transformation de message est complexe à réaliser, il est possible d'enchaîner les translateurs afin que chacun d'eux effectue une partie de la transformation. Un exemple d'enchaînement de translateurs et du résultat peut être vu à la figure 4.6

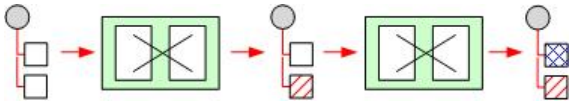

FIG. 4.6 – Pictogramme : enchaînement de *Translator*s

4.5.2 Modélisation

Comme le *MessageChannel* et le *Router*, le *Translator* est une spécialisation du composant de base et est modélisé par une classe abstraite. Il possède un point de réception des messages et envoie tous les messages produits vers un seul composant destinataire. Une contrainte appliquée à ce composant est que chaque message reçu doit donner lieu à un message émis.

La quantité de messages stockés temporairement est configurée comme dans les *Channel*s et *Router*s même si cette valeur devrait être égale à "1" selon [**Hohpe**]. Nous préférons laisser ces paramètres configurables afin d'augmenter la souplesse du traitement des messages.

4.5.3 Exemple

Un message, reçu par le translateur, pourra par exemple contenir sous forme de texte le numéro d'immatriculation d'un véhicule et devra être tran-

formé en message XML :

- Message reçu :
  `FUN123`
- Message réémis :
  `<PlaqueImmatriculation>GFZ334</PlaqueImmatriculation>`

4.6 Message Endpoint

4.6.1 Présentation

Le *Message Endpoint*, représenté par le pictogramme de la figure 4.7, est le composant chargé de l'interface entre une application et le système. Selon [**Hohpe**], il permet aux yeux d'une application d'encapsuler le système de messagerie.

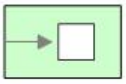

FIG. 4.7 – Pictogramme : Message Endpoint

Deux types de *Endpoint* existent :

- le *Endpoint* client d'une application. Il transforme les informations à envoyer vers le système en un message qu'il envoie vers un composant du système : le *Endpoint* d'entrée ;
- le *Endpoint* fournisseur d'une application. Il reçoit du système un message, l'analyse, en extrait les informations et les transmet à l'application : le *Endpoint* de sortie.

La communication avec un *Endpoint* peut se faire selon deux modes différents :

- le *Endpoint* client va chercher l'information directement dans le système d'information de l'application. Il doit alors utiliser un mécanisme de type *Timer* pour détecter ou non la présence d'informations à transmettre. On parle de *Polling Consumer* ;
- le *Endpoint* client reçoit l'information de l'application. Il agit dès qu'il reçoit cette information. On parle alors de *Event-Driven Consumer*.

Ces modes de communication sont également d'application pour un *Endpoint* fournisseur où le mode *Polling* consiste à fournir l'information à la demande de l'application et le mode *Event-Driven* implique que le *Endpoint* fournisse l'information dès la fin du traitement d'un message reçu du système.

Deux types d'applications peuvent être rencontrés :
- l'application ne peut communiquer avec une autre de part son ancienneté ou son design. Dans ce cas, le *Endpoint* fonctionnera en mode *Polling Consumer* lorsque le *Endpoint* sera client et en mode *Event-Driven Consumer* pour un *Endpoint* fournisseur. Nous reviendrons plus en détail sur ces deux modes dans le chapitre consacré aux *Endpoint*s.
- l'application peut être modifiée pour permettre de dialoguer avec un autre système. Dans ce cas, le mode de fonctionnement dépendra des accords convenus entre les responsables de l'application et ceux en charge du *Endpoint*.

4.6.2 Modélisation

Un *Endpoint* est, à ce niveau de l'étude, modélisé par une classe abstraite. Cette classe utilise une couche spécifique à l'application avec laquelle il dialogue. Cette couche est modélisée par une classe spécifique à cette application. L'implémentation exacte de cette couche devra être spécifiée lors de la description du composant au niveau du DSL.

Le type de client est matérialisé par les attributs *isClient* et *isProvider*. Le mode de communication est indiqué par la valeur de l'attribut *communicationMode*.

4.6.3 Exemple

Nous utilisons dans notre exemple un *Endpoint* pour permettre à l'application utilisée par l'Agent de Police l'envoi de constats d'excès de vitesse.

4.7 Modélisation du système de messagerie et du composant

4.7.1 Système de messagerie

Le système de messagerie que nous voulons modéliser a pour but l'intégration d'applications via l'échange d'informations. Cet échange se fait sous la forme de messages asynchrones. Le dialogue entre applications via le système peut être représenté par la figure 4.8.

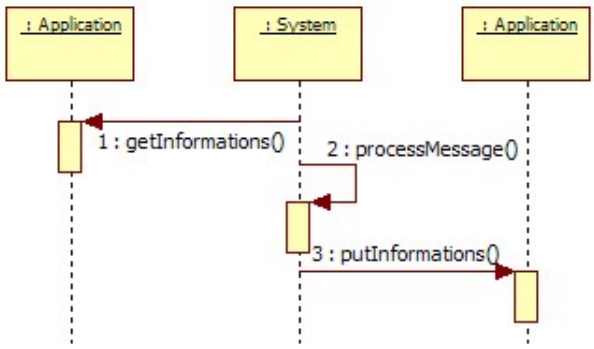

FIG. 4.8 – Diagramme de séquence : Dialogue entre applications et système

Remarque : ce schéma montre d'une part que le *Endpoint* client en mode *PollingConsumer* tandis que le *Endpoint* fournisseur fonctionne en mode *Event-Driven Consumer*. Représenter d'autres modes implique de changer les modes de communication de ces *Endpoint*s.

Chaque information, pour être transmise d'une application à une autre, est transformée en un message. Ce message est transporté de composant en composant à travers le système. Enfin, ce message est reformaté en termes compréhensibles par l'application destinatrice par un *Endpoint*. Lorsque le système achemine une demande puis une réponse, les deux messages qui en découlent sont transportés sous la forme de deux messages indépendants pour le système.

L'information à transmettre est captée par le système via un *endpoint* qui l'encapsule dans un message. Ce message est transmis à un *composant* interne. Ensuite, de *composant* interne en *composant* interne, le message est acheminé vers un *endpoint* qui délivre l'information à l'application destinataire après avoir analysé le message reçu.

4.7.2 Composant du système

Les éléments qui constituent le système peuvent dès lors être rangés dans l'une des deux catégories suivantes :

- les composants à la frontière du système : les *Endpoint*s et
- les composants internes que nous appellerons *InternalComponent*s.

Nous pouvons dès à présent poser la contrainte suivante : un *Endpoint* ne peut pas dialoguer directement avec un autre *Endpoint* du même système. Ceci rendrait inutile l'usage du système au niveau transport asynchrone de messages bien que cela permettrait un transfert d'informations entre deux applications qui ne savent pas communiquer.

4.7.3 Transport de messages entre composants

Avant de nous pencher sur la structure d'un composant, voyons comment sont transmis les messages entre deux composants. Certaines de ces informations ne se trouvent pas dans le DSL mais peuvent aider à la compréhension du système et à l'utilisation du DSL.

Réception d'un message

Comme un *Component* peut avoir un nombre encore indéterminé de points de réception de messages à ce moment de l'étude, nous décidons de modéliser chaque "fonction chargée de la réception des messages", qui matérialise un point de réception, par une classe. Cette classe est appelée un *Port* par analogie aux différents points de connexion d'un routeur au sein d'un réseau. De cette façon, lorsque nous devrons ajouter un port à un composant, il suffira de lui ajouter une instance de cette classe.

Chaque port possède une méthode unique : *send()*. L'appel de cette méthode par le composant qui souhaite transmettre un message à son correspondant se fait selon un mode synchrone : le composant émetteur appelle la méthode *send()* d'un port du composant récepteur et attend que celui-ci lui réponde avant de continuer son traitement. Lorsque le port reçoit par cette méthode un message, il le stocke dans la mémoire du composant afin de permettre son traitement ultérieurement.

Au sein d'un composant, la zone mémoire, modélisée par la classe *Memory*, est composée de deux parties :

- la zone *IncomingMessages* qui est utilisée par les ports pour stocker les messages à traiter par le composant et
- la zone *OutgoingMessages* utilisée elle pour le stockage temporaire des messages à acheminer vers les destinataires.

La taille maximale de ces zones est définie par configuration par les attributs *maxIncomingMessages* et *maxOutgoingMessages* que nous avons rencontrés plus tôt.

Traitement d'un message

Le traitement proprement dit d'un message reçu par le composant est confié à une classe dite *MessageProcessor*. Cette classe traite un message reçu et place le message à envoyer dans la zone *OutgoingMessages*. Toute spécialisation du composant devra dès lors avoir une spécialisation correspondante du *MessageProcessor*. Pour certains composants, le rôle de ce *MessageProcessor* est clairement défini ; il ne sera pas nécessaire de le configurer dans ce cas. Cette donnée sera obligatoire pour d'autres types de composants.

Ainsi, transmettre un message reçu à tous les destinataires est évident. Par contre, utiliser un algorithme de sélection de destinataires en fonction de status d'autres composants peut être beaucoup moins standard tout comme la transformation de format de message.

Envoi d'un messge

Pour envoyer un message à son correspondant, le composant émetteur utilise une référence vers le port à utiliser chez le composant récepteur. L'envoi du message, fonction particulière d'un composant, est confiée à une nouvelle classe appelée *Sender*. Le rôle de cet élément est de consulter la zone *OutgoingMessages* du composant pour obtenir un message à envoyer et de l'envoyer vers son destinataire.

Dialogue

Le dialogue entre deux composants peut être modélisé à l'aide du diagramme de la figure 4.9.

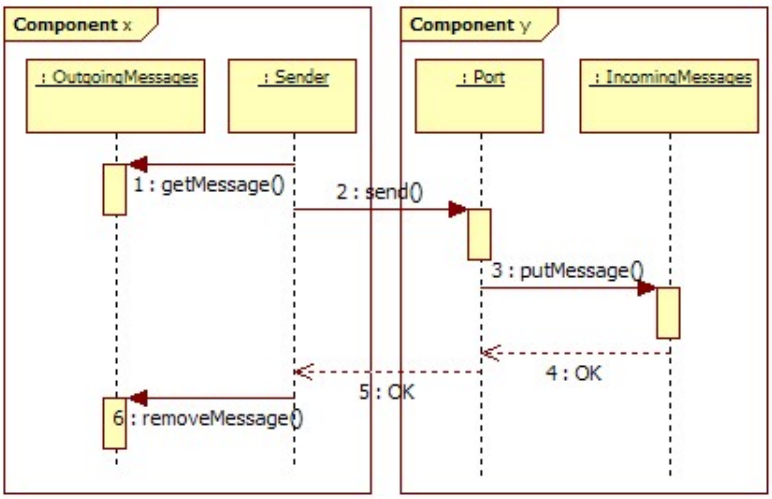

FIG. 4.9 – Diagramme de séquence : dialogue entre composants

4.7.4 Principaux états d'un composant

Les états d'un composant peuvent alors être représentés à l'aide du diagramme de la figure 4.10.

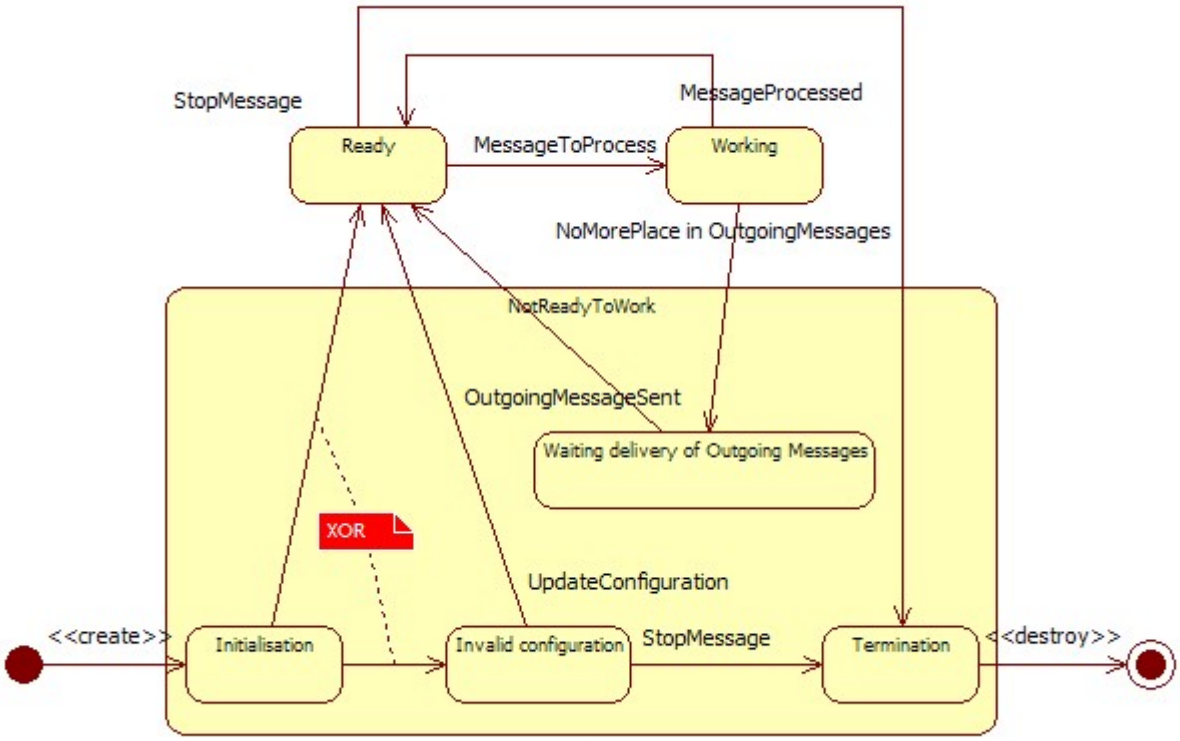

FIG. 4.10 – Diagramme d'états : Etats d'un composant

Chaque port d'un composant aura également deux états :
- Ready : lorsque le port peut accepter un message et
- NotReady : lorsque le composant qui le possède est en état "NotReadyToWork" ou lorsque la capacité de stockage de messages entrants est atteinte.

4.7.5 Composition de composants ou *Processus*

Nous n'avons pas encore rencontré ce cas dans les composants proposés par [**Hohpe**] mais au vu de notre système de messagerie, nous pouvons raisonnablement poser que ce système pourra également être composé de processus.

Chaque processus sera modélisé par une instance de la classe *Process*. Il est composé d'un assemblage de composants simples comme ceux que nous avons vus jusqu'à présent. Un processus peut avoir des ports à condition que chacun de ceux-ci soit relié à un port d'un des composants internes au processus. L'envoi des messages vers l'extérieur du processus sera confié aux composants qui le composent.

Remarque : le port d'un processus transmet les messages qu'il reçoit au port du composant interne dont il a la référence. Les deux types de ports présentent la même interface et ont un rôle très similaire, ils sont donc modélisés à l'aide du diagramme de la figure 4.11.

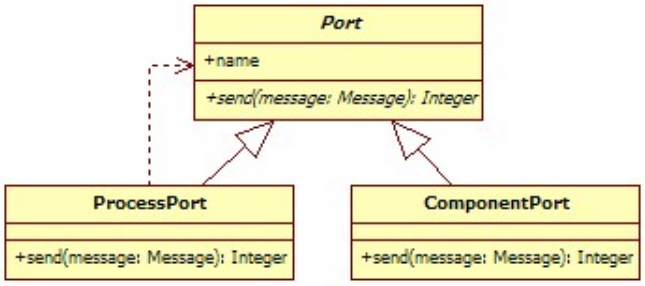

FIG. 4.11 – Diagramme de classes : Ports

4.7.6 Hiérarchie des composants

Nous devons dès lors distinguer les processus des composants plus élémentaires que nous avons introduits plus tôt pour aboutir à la hiérarchie des composants que nous avons rencontrés à la figure 4.12.

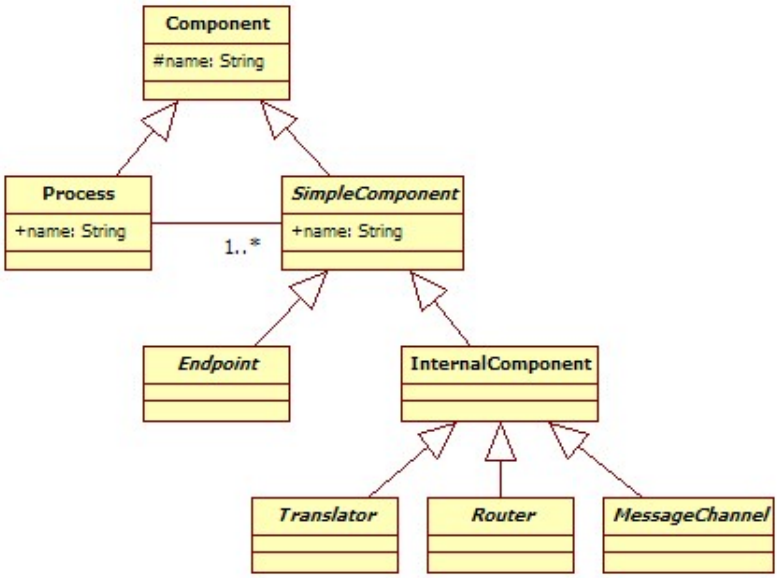

FIG. 4.12 – Diagramme de classes : Hierarchie des composants

4.7.7 Structure et composition des composants de base

SimpleComponent

Le modèle interne du composant simple peut alors être présenté sous la forme du diagramme de classes de la figure 4.13.

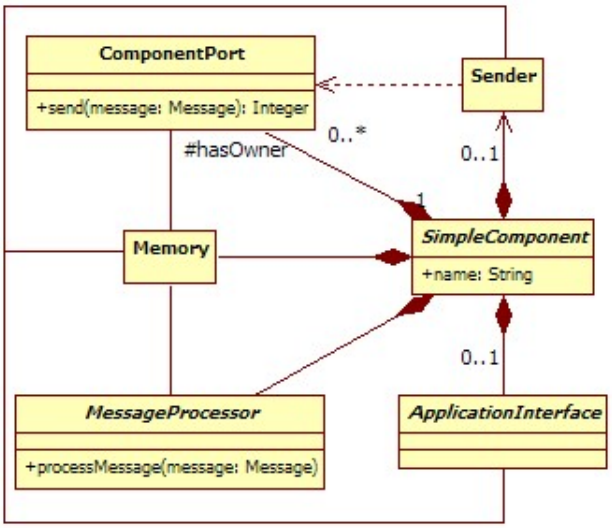

FIG. 4.13 – Diagramme de classes : structure d'un composant

où nous voyons que le composant à modéliser peut être composé des modules suivants :

- *MessageProcessor* chargé du traitement des messages ;
- *Memory* responsable du stockage des messages reçus et à envoyer ;
- un *ApplicationInterface* utile pour les endpoints ;
- quelques ports et enfin
- un émetteur de messages : le *Sender*.

Remarque : ce diagramme est utile pour l'opérationnel mais permet de mettre en évidence le fait que certains composants devront être dotés d'un de ces éléments. Nous préciserons cela en temps voulu.

Chapitre 5

Canaux de communication

5.1 Introduction

Dans les chapitres précédents, nous avons introduit le message ainsi que le composant de base et la classification proposée par [**Hohpe**]. Nous avons également proposé une modélisation du composant de base. Nous allons maintenant entrer dans les détails spécifiques à chaque catégorie de composants proposés en commençant par le *Message Channel*.

La spécialisation du *Channel* se fait sur base des critères suivants :
- le nombre de destinataires ;
- le contrôle du type de message acheminé ;
- le traitement de message invalide ou sans destinataire ;
- la persistance des messages en cas d'interruption du système de messagerie (panne de courant, ...) ;
- le transfert de messages d'un système de messagerie à un autre système.

5.2 Point-to-Point channel

5.2.1 Présentation

Le *Point-to-Point Channel*, représenté par le pictogramme de la figure 5.1 est l'implémentation la plus directe du *Message Channel* décrit dans le chapitre 4.3. Il possède un seul port et transmet tous les messages reçus vers

un seul destinataire.

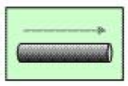

FIG. 5.1 – Pictogramme : point-to-point channel

Le traitement de ce *Channel*, consiste à placer chaque message reçu dans la liste des messages à envoyer vers son unique destinataire.

5.2.2 Modélisation

Le *Point-to-Point Channel* est l'implémentation la plus directe du *Message Channel*.

Remarque : Le destinataire d'un message étant en fait un port d'un composant, nous pouvons modéliser ce destinataire comme la référence d'un port d'un composant. Nous introduisons donc une nouvelle classe dans notre modèle : la classe *Destination* qui contient une référence vers un port.

5.2.3 Contraintes

- le *Point-to-Point channel* doit avoir un seul port d'entrée et un seul destinataire :
  ```
  Context PointToPointChannel
       inv: destinations->size() = 1
  ```
- le destinataire ne peut référencer le port d'entrée pour transmettre les messages (pour éviter le bouclage) ;
- chaque message reçu sur le port d'entrée doit être expédié vers le destinataire.

5.2.4 Exemple

Le channel qui transporte les messages reçus du radar les envoie uniquement vers la Centrale des Infractions.

5.3 Publish-Subscribe Channel

5.3.1 Présentation

Le *Publish-Subscribe Channel*, représenté par le pictogramme 5.2, est le deuxième *Channel* décrit en fonction du nombre de destinataires : ce *Channel* admet plusieurs destinataires qui devront recevoir tous les messages reçus.

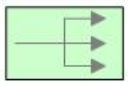

FIG. 5.2 – Pictogramme : publish-subscribe channel

Le traitement à réaliser par ce *Channel* consiste à placer dans la zone des messages à délivrer autant de copies du messages à envoyer qu'il y a de destinataires définis pour cette instance du composant. Chaque copie du message est associée à un destinataire.

Ce traitement étant très semblable à celui décrit pour le *Channel* précédant, nous décidons de généraliser le *ChannelMessageProcessor*. Ainsi, au lieu de prévoir une seule zone pour mémoriser le destinataire au sein du *Point-to-Point Channel*, nous décidons de placer dans la zone mémoire des *Channel*s (classe *Memory*), une liste des destinataires possibles pour un *Channel*. Cette liste ne pourra contenir qu'un seul élément dans le cas du *Point-to-Point*.

5.3.2 Modélisation

Comme le traitement de ce *Channel* est similaire au traitement réalisé par le *PointToPointChannel*, nous modélisons le *Publish-Subcribe* comme un *PointToPointChannel* sans la contrainte de limitation du nombre de destinations.

5.3.3 Contraintes

- le port d'entrée ne peut être référencé par un des destinataires pour éviter un bouclage ;
- deux destinataires ne peuvent référencer un même port pour éviter la duplication des messages ;
- un message reçu doit être publié vers tous les destinataires connus du *Channel*.

5.3.4 Exemple

En nous basant sur notre exemple, on utilise un *Publish-Subscribe Channel* pour envoyer les informations de l'excès de vitesse d'une part à la D.I.V. et d'autre part au Tribunal de Police.

5.4 Datatype Channel

5.4.1 Présentation

Le *Datatype Channel* représenté par le pictogramme de la figure 5.3 assure au destinataire du *Channel* que les messages qu'il va recevoir sont d'un type bien précis et uniquement de ce type. Ce *Channel* agit donc comme un filtre sur base du type de message.

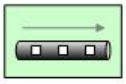

FIG. 5.3 – Pictogramme : Datatype channel

Le module de traitement va vérifier que le message reçu correspond au type demandé. Lorsque le message est valide au niveau du type, le message est transmis au destinataire du *Channel*. Lorsque le message n'a pas le type attendu, le message est considéré comme invalide et est envoyé vers un *Invalid Message Channel* que nous détaillerons plus loin dans ce travail.

5.4.2 Modélisation

Le *DataType Channel* possède comme attribut le nom du type de message accepté. Il peut également connaître le port d'entrée d'un *Invalid Message Channel*.

5.4.3 Contraintes

- le composant doit avoir un seul destinataire ;
  ```
  Context PointToPointChannel
       inv: destinations->size() = 1
  ```
- le destinataire ne peut référencer le port afin d'éviter le bouclage du message ;
- le message reçu peut être transmis uniquement si le type du message est celui défini ;

– le *MessageProcessor* doit disposer d'une instance de *MessageTypeFilter* valide, faute de quoi, il n'accepte aucun message.

5.5 Invalid Message Channel

5.5.1 Présentation

Comme nous l'avons vu au paragraphe précédent, il nous faut prévoir un mécanisme pour traiter les messages non autorisés c'est-à-dire qui ne sont pas acceptés par un *Datatype Channel* par exemple. Ceci est le rôle de l'*Invalid Message Channel* dont la représentation nous est fournie par le pictogramme de la figure 5.4.

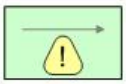

FIG. 5.4 – Pictogramme : Invalid Message Channel

Ce *Channel* est utilisé pour retirer du système de messagerie les messages invalides ou incorrects. Envoyer un message dans ce type de *Channel* revient pratiquement à le détruire. L'un des avantages d'utiliser un tel *Channel* est de pouvoir tenir des statistiques de messages de ce type en appliquant un traitement lors de la réception de ceux-ci. Un autre avantage pourra être de récupérer certains messages qui auraient été détruits en connectant, à fin d'analyse, un destinataire au cours de son cycle de vie : les messages pourraient alors être analysés pour éventuellement localiser les conditions d'invalidité de ces messages.

Lorsque ce *Channel* ne dispose pas d'une destination pour les messages qu'il reçoit, il les détruit.

5.5.2 Modélisation

Ce *Channel* est une spécialisation du *MessageChannel*.

5.5.3 Contraintes

– le *Channel* dispose d'un port d'entrée et d'au plus un destinataire ;

```
Context PointToPointChannel
     inv: destinations->size() <= 1
```

– lorsque le *Channel* dispose d'un destinataire, les messages stockés sont envoyés vers ce destinataire ;

– dans le cas contraire, la conservation des messages dépend de la configuration du *Sender* et du temps passé par le message dans la zone *OutgoingMessages*.

5.5.4 Exemple

Un *Invalid Message Channel* pourrait par exemple être utilisé lorsqu'on envoie un *Document Message* à un composant destiné à recevoir des *Event Message*. On pourra également utiliser ce type de *Channel* dans le cas de messages au format XML, le *Channel* pourrait rejeter le message XML qui n'est pas bien formé.

5.6 Dead Letter Channel

5.6.1 Présentation

Nous allons envisager maintenant le cas où un message ne peut être transmis à son destinataire. Cela peut se produire lorsque le port de destination a été mal configuré ou que ce port n'existe pas ou plus. [**Hohpe**] introduit à cet effet le *Dead Letter Channel* représenté par le pictogramme de la figure 5.5.

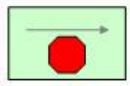

FIG. 5.5 – Pictogramme : Dead Letter Channel

5.6.2 Modélisation

Ce *Channel* est modélisé exactement comme un *Invalid MessageChannel*. Afin d'insister sur la différence de concept entre un message invalide et un message n'ayant plus de destinataire valide, nous modélisons ces deux classes comme des spécialisations de la classe abstraite *TrashChannel* qui implémentera le code commun aux deux types de *Channel*s.

5.7 Guaranteed Delivery

5.7.1 Présentation

Jusqu'à présent, dans notre parcours sur les spécialisations du *Message Channel*, nous avons vu comment contrôler le nombre de destinataires et comment retirer du système de messagerie les messages invalides ou qui n'ont plus de destinataires. Nous allons maintenant nous concentrer sur la façon de rendre plus robuste un *Channel* de façon à supporter une panne de courant ou une avarie dans le réseau : le *Guaranteed Delivery* dont le pictogramme est repris à la figure 5.6.

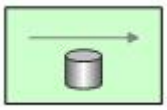

FIG. 5.6 – Pictogramme : Guaranteed Delivery

Ce *Channel* stocke de façon persistante les messages qu'il reçoit et ceux qu'il doit expédier. Ceci permet une tolérance aux pannes : lorsque le système tombe en panne, le *Channel*, à son redémarrage retrouve les messages à traiter ou à envoyer avant l'arrêt du système et reprend son traitement. Il ne faudra pas oublier de retirer de la mémoire persistante les messages déjà traités ou envoyés. Il doit être possible de désactiver ce stockage pendant les phases de corrections d'erreurs ou de tests.

5.7.2 Modélisation

Comme ce système de stockage peut être utilisé par tous les types de *Channel*s, nous ne modéliserons pas ce composant sous une forme particulière mais plutôt comme une option offerte aux *Channels*. Nous ajoutons donc à tous les *Channel*s et donc à leur classe abstraite l'attribut booléen *mustPersistData* et un attribut *PersistanceData* facultatif qui est utilisé pour la persistance des données vers par exemple une base de données.

5.8 Channel Adapter

5.8.1 Présentation

Le *Channel Adapter* représenté par [**Hohpe**] par le pictogramme de la figure 5.7 est un composant qui fait la liaison entre une application et le

système de messages comme un *Endpoint*. Il accède aux données de l'application sans que celle-ci soit modifiée pour pouvoir être intégrée à d'autres applications.

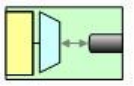

FIG. 5.7 – Pictogramme : Channel Adapter

Etant donné qu'il s'apparente plus par son rôle à un *End point*, nous reviendrons sur ce composant au chapitre qui traite de ces composants.

5.9 Messaging Bridge

5.9.1 Présentation

Le *Messaging Bridge*, représenté par le pictogramme de la figure 5.8, a pour rôle la transmission de messages d'un système de messagerie à un autre système. Il achemine les messages qu'il reçoit sur un port vers un destinataire qui se trouve dans un autre système de messagerie.

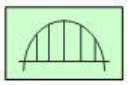

FIG. 5.8 – Pictogramme : Messaging Bridge

Ce composant est le résultat de l'assemblage de deux *Channel Adapter*s où les partenaires extérieurs sont en fait deux systèmes de messagerie. Ceci motive le fait que le *Channel Adapter* ait été rangé dans la catégorie des *Channel* et pas dans celle des *EndPoint*s par [**Hohpe**].

Comme le *Messaging Brigde* est une encapsulation de deux *Channel Adapter*s, nous choisissons de ne pas l'implémenter car nous ne voyons pas la valeur ajoutée apportée par un tel composant à part le fait d'occulter le transport de message d'un système à l'autre ce qui pourrait nuire à la clarté des composants installés au sein d'un système de messagerie.

5.10 Message Bus

5.10.1 Présentation

Le *Message Bus*, représenté par le pictogramme de la figure 5.9, est un assemblage de composants plus élémentaires. Ce composant reçoit des messages sur plusieurs ports d'entrée et selon les *Component*s qui le composent, il achemine ces messages vers un(plusieurs) port(s) de sortie.

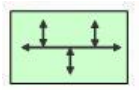

FIG. 5.9 – Pictogramme : Message Bus

5.10.2 Modélisation

Un *Bus* est modélisé comme un processus. Il sera fonction de l'assemblage de composants qui le constitue.

5.10.3 Exemple

Nous pouvons par exemple avoir un *Message Bus* chargé du traitement des commandes qu'il reçoit. La commande sous la forme d'un message est déposée sur l'un des ports du bus. Ensuite le message est envoyé d'une part vers la facturation d'une entreprise et d'autre part vers le service de livraison par l'utilisation d'un *Publish-Subscribe Channel*. Ensuite, chacun de ces départements répond en envoyant les informations appropriées sur un channel vers un composant (encore à préciser et spécifier) qui corrèle les informations obtenues. Le résultat final étant envoyé par le bus vers une application via un port de sortie du bus.

5.11 Remarques à propos des *Message Channel*s

5.11.1 Paramètres qui caractérisent les différents types de *Channel*s

1. Nombre de destinataires :

Channel	Destinataire(s)	
	minimum	maximum
Point-to-Point	1	1
Publish-Subscribe	1	pas de limite
DataType	1	1
Invalid Message	0	1
Dead Letter	0	1

2. Persistance des messages reçus et échangés ;
3. Connaissance d'un *Invalid Message Channel* ;
4. Connaissance d'un *Dead Letter Channel* ;

Nous avons placé dans cette liste, sans le mentionner explicitement jusqu'à présent, le dernier paramètre de cette liste. Ce paramètre, facultatif, est utilisé pour l'expédition des messages lorsque le destinataire du message ne répond plus. Cette situation peut se produire lorsque le destinataire est arrêté par exemple. Quand ce paramètre n'est pas défini, le message est détruit.

5.11.2 Remarque à propos du *Guaranteed Delivery*

Nous avons modélisé ce pattern comme une option des *Channel*s, implémentée selon une interface. Or nous avons modélisé tous nos *SimpleComponent*s avec une mémoire pour enregistrer notamment tous les messages reçus et à envoyer. Nous pouvons dès lors généraliser cette option à tous les *SimpleComponent*s et pas seulement aux *Channel*s.

Le déplacement de l'interface *IGuaranteedDelivery* vers la classe *SimpleComponent* implique également le déplacement de l'attribut *mustPersistData*.

Chapitre 6

Construction de messages

6.1 Introduction

Après avoir proposé une modélisation des *Message Channel*s, penchons-nous maintenant avec plus d'attention sur les messages en insistant sur les types proposés par [**Hohpe**] ainsi que sur les éléments qui composent le Header.

Remarque : encore une fois, les messages n'apparaissent pas dans le DSL que nous voulons modéliser mais ils sont utiles à la compréhension du fonctionnement de notre système d'intégration.

6.2 Command Message

6.2.1 Présentation

Le *Command Message*, représenté par le pictogramme de la figure 6.1 est le premier type de message proposé par [**Hohpe**]. Il correspond à un ordre envoyé à une application. La réception de ce message a pour but de déclencher une action de la part du récepteur.

FIG. 6.1 – Pictogramme : Command Message

6.2.2 Modélisation

Le *Command Message* sera matérialisé par une spécialisation de la classe *Message*. Ce message est composé d'un nom de commande et d'une liste de paramètres. Chaque paramètre, *MessageParameter*, sera modélisé comme une classe composée de deux attributs : un nom et une valeur.

6.2.3 Exemple

Comme nous n'avons pas présenté ce type de message dans notre exemple, nous reprenons l'exemple proposé par [**Hohpe**] : ce message pourrait être "setNewPrice", le nom de la commande, suivi du nouveau prix et de la date d'application de ce prix.

6.3 Document Message

6.3.1 Présentation

Le *Document Message*, représenté par le pictogramme de la figure 6.2 est destiné à acheminer des informations sous la forme d'un texte libre, au format XML ou structuré ...

FIG. 6.2 – Pictogramme : Document Message

6.3.2 Modélisation

Le *Document Message* contient un ensemble d'informations que le composant récepteur doit traiter. Comme ce texte pourra prendre plusieurs formes, nous poserons simplement que le contenu de ce message est un flot de caractères.

6.3.3 Exemple

Un exemple de ce type de message peut être le message qui contient les données du propriétaire d'un véhicule : son nom, ses prénoms, son adresse.

6.4 Event Message

6.4.1 Présentation

Ce type de message, représenté par le pictogramme de la figure 6.3, est un message destiné à signaler la survenance d'un événement.

FIG. 6.3 – Pictogramme : Event Message

6.4.2 Modélisation

Nous modélisons ce type de message comme nous avons modélisé le *CommandMessage* : une spécialisation du *Message* et une liste de paramètres décrivant l'événement.

6.4.3 Exemple

Un exemple de ce type de message est le message que le radar envoie vers la Centrale des Infractions lorsqu'il détecte un excès de vitesse.

6.5 Request-Reply Message

6.5.1 Présentation

Dans ce chapitre, nous évoquons le mécanisme de "Requête/Réponse" en rappelant que nous avons envisagé jusqu'ici le mécanisme d'envoi de messages "One-way" : un message est envoyé à un destinataire ; l'émetteur de ce message continue son processus dès la fin de l'envoi. Il recevra une réponse de façon asynchrone.

[**Hohpe**] décrit ce que pourrait être l'envoi d'un message (requête) et l'attente d'une réponse à ce message avant la reprise du processus en cours. Pour ce faire, il envisage le blocage du processus au sein de l'élément émetteur tant qu'il ne reçoit pas de réponse à son message initial. Ce type de message

est représenté par le pictogramme de la figure 6.4.

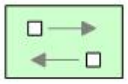

FIG. 6.4 – Pictogramme : Request-Reply

6.5.2 Modélisation

Nous modélisons cette information par une spécialisation du type de *Header* : le *RequestReplyHeader*. Lorsqu'un composant envoie un message avec ce type de *Header*, il suspend tout son processus jusqu'à ce qu'il ait reçu le message de réponse.

Il est important de noter qu'un message contenant ce type de *Header* devra être traité en priorité par les composants chargés d'acheminer un message à travers le système ou de répondre à la requête.

6.6 Return Address

6.6.1 Présentation

Le *Return Address* est un mécanisme qui permet la réalisation du mécanisme de *Request-Response* : deux émetteurs différents créent chacun une requête qui est publiée sur le même *Channel*. Pour permettre la réception de la réponse à leur requête, ils placent dans un *Header* une adresse qui est la référence d'un de leurs ports de réception.

Lorsque le composant chargé de la réponse émet le message, il utilise cette information pour envoyer la réponse au destinataire indiqué.

6.6.2 Modélisation

Nous devons donc définir une spécialisation du *Header*, le *ReturnAddressHeader*, pour envoyer l'adresse de retour dans le message de requête.

6.7 Correlation Identifier

6.7.1 Présentation

Comme extension du *Return Address*, nous avons un second mécanisme à utiliser pour les requêtes-réponses. Ce mécanisme consiste à placer dans le message réponse un identifiant qui permet de relier la requête à la réponse. Le *Correlation Identifier* est représenté par le pictogramme de la figure 6.5.

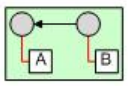

FIG. 6.5 – Pictogramme : Correlation Identifer

L'un des moyens d'identifier la réponse à une requête est de placer dans cet *Header* l'identifiant du message de la requête.

6.7.2 Modélisation

Nous allons dès lors modéliser le *Correlation Identifier* comme un élément du header du message c'est-à-dire comme une spécialisation d'un *Header* : le *CorrelationIdentifierHeader*.

6.8 Message Sequence

6.8.1 Présentation

Lorsque la réponse ou simplement un message est trop grand, il est parfois utile de transmettre le message en plusieurs parties. L'auteur du livre cite notamment certaines limitations quant à la taille des messages échangés en COBOL ou par certains mainframes. Il propose dès lors d'utiliser un numéro de séquence et représente ce type d'information à l'aide du pictogramme de la figure 6.6.

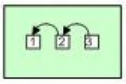

FIG. 6.6 – Pictogramme : Message Sequence

Il existe deux mécanismes d'identification d'éléments d'une séquence. Soit

on indique pour chaque élément de séquence le nombre d'éléments dans la séquence soit on utilise un drapeau pour indiquer qu'un élément de séquence est le dernier de la séquence.

6.8.2 Modélisation

Nous modélisons cette information dans un premier temps par la classe *SequenceIdentifierHeader* qui contient l'identifiant de la séquence et le numéro de l'élément au sein de la séquence. Nous définissons ensuite deux sous-classes concrètes *SizeSequenceIdentifierHeader* et *FlagSequenceIdentifierHeader*. La première aura un attribut qui indique le nombre d'éléments dans la séquence, la seconde possèdera un attribut booléen qui indiquera quand on manipule le dernier élément de la séquence.

6.9 Message Expiration

6.9.1 Présentation

Il nous faut ensuite envisager le traitement qu'il faut apporter à un message qui a "traîné" dans un système de messages où l'un des composants a par exemple été longtemps indisponible. Le principe de ce mécanisme consiste à placer dans l'en-tête du message un "timestamp". Ce timestamp indique lorsqu'il est présent jusque quand il faut traiter un message. Tout message qui est expiré doit alors être rejeté dans un *Dead Letter Channel* lorsqu'il y en a un.

6.9.2 Modélisation

La modélisation de cet élément se fera de façon analogue aux autres éléments d'en-tête décrits dans ce chapitre c'est-à-dire une spécialisation de l'en-tête : le *MessageExpirationHeader*. Cet élément contient une date (et heure) indiquant la fin de la validité du message.

6.10 Format Indicator

6.10.1 Présentation

Après avoir envisagé une classification des messages et certains attributs, nous devons envisager le cas où le format d'un message doit évoluer : au cours de l'histoire des composants, il peut arriver qu'il faille ajouter certaines

données à un message ou en changer le format de ce message. Nous plaçons un indication sur le format d'un message dans un élément du *Header* comme nous l'avons fait pour les cas précédents. Grâce à cette indication de format, il devient possible de faire évoluer les messages et les composants qui traitent ce message.

6.10.2 Modélisation

Nous allons modéliser cet attribut du message comme une spécialisation de l'en-tête : la classe abstraite *FormatIndicatorHeader*. Nous pourrons spécialiser ce *Header* afin d'y placer selon les besoins un numéro de version, un identifiant significatif ou même des informations de type XMLSchema afin de valider un message XML.

6.11 Synthèse concernant la construction des messages

Dans ce chapitre, nous avons modélisé les trois types de messages proposés en tant que spécialisations du message. Nous présentons à l'aide du diagramme de classes de la figure 6.7 la spécification complète du *CommandMessage*.

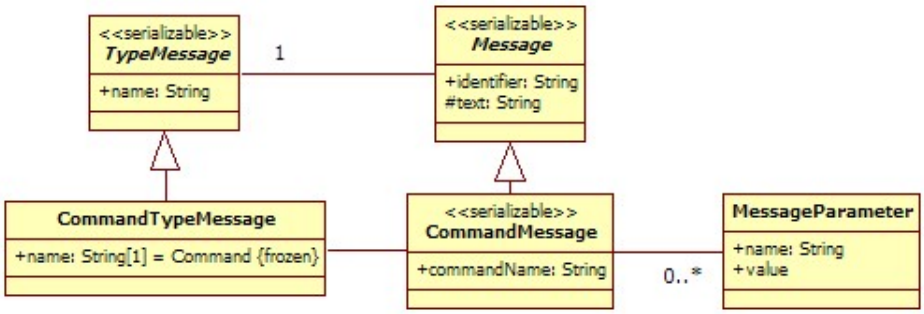

FIG. 6.7 – Diagramme de classes : CommandMessage

Nous pouvons représenter les différents éléments d'en-tête du message sous la forme du diagramme de la figure 6.8.

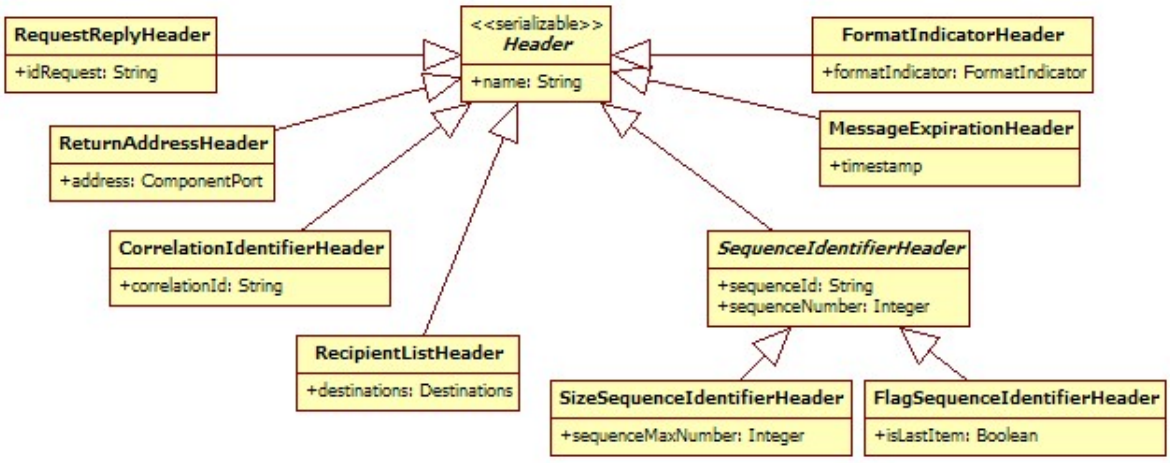

FIG. 6.8 – Diagramme de classes : spécialisation du Header

Chapitre 7

Routage des messages

7.1 Content-Based Router

7.1.1 Présentation

Dans cette partie, nous allons affiner les composants chargés du routage des messages évoqués plus avant dans ce travail dans la partie 4.4. Le pictogramme proposé par ce composant est le même que celui du routeur décrit dans la partie 4.

Le *Content-Based Router* achemine les messages reçus sur base de leur contenu : l'algorithme de choix du destinaire est codé dans le *Router*. L'algorithme de sélection est spécifique à chaque implémentation de cette classe. Le nombre de destinataires requis par l'algorithme de sélection est indiqué dans l'attribut *RequiredDestinations* dont la valeur est fixée. Lorsque le nombre de destinations configurées n'est pas celui requis, le composant est dans l'état Invalid et ne peut accepter aucun message.

7.1.2 Modélisation

Le *Content-Based Router* est modélisé par une spécialisation de la classe *Router*.

7.2 Message Filter

7.2.1 Présentation

Le *Message Filter* est un routeur qui transfère un message à son destinataire uniquement s'il correspond à certaines conditions. Le message qui ne correspond pas à la (aux) condition(s) définie(s) dans le routeur est envoyé vers un *Invalid Message Channel* lorsqu'un tel type de *Channel* est associé au composant.

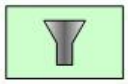

FIG. 7.1 – Pictogramme : Message filter

Parmi les types de conditions qui peuvent être définies, [**Hohpe**] explique que le *Message Filter* peut servir à éliminer les messages dupliqués : le *Filter* enregistre les messages qu'il a déjà transmis et élimine les messages identiques qu'il reçoit après la première transmission de ceux-ci.

Nous remarquons que ce composant agit d'une façon très semblable à celle décrite pour le *Datatype Channel* : il possède un point d'entrée et un destinataire à qui il transmet les messages valides. Le choix d'un destinataire ne semble pas la fonction principale de ces deux composants ; choix qui n'est même pas évoqué pour le *Datatype Channel*.

Nous décidons donc de retirer ces deux composants de la catégorie dans laquelle ils se trouvent pour être placés dans une nouvelle catégorie : *Filter*.

7.2.2 Modélisation

Nous modélisons la nouvelle catégorie par une classe abstraite, *Filter*, qui possède une méthode *acceptMessage()*. Les classes *Datatype Filter* et *MessageFilter* sont des spécialisations de cette classe.

7.2.3 Contraintes

- le *Point-to-Point channel* doit avoir un seul port d'entrée et un seul destinataire :
  ```
  Context PointToPointChannel
       inv: destinations->size() = 1
  ```
- le destinataire ne peut référencer le port d'entrée pour transmettre les messages (pour éviter le bouclage) ;

7.3 Dynamic Router

7.3.1 Présentation

Dans certains cas, la configuration statique du choix du destinataire d'un message n'est pas suffisante pour les besoins du système car certains composants peuvent être surchargés ou traiter les messages moins rapidement que prévu. Le *Dynamic Router*, représenté par le pictogramme de la figure 7.2 peut être configuré dynamiquement.

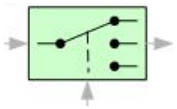

FIG. 7.2 – Pictogramme : Dynamic Router

La sélection du port de destination se fait sur base d'informations fournies par les destinataires possibles d'un message. Ces informations, stockées dans le *Router*, sont susceptibles d'évoluer au cours de la vie de ce composant : un destinataire peut transmettre de nouvelles informations suite par exemple au temps de traitement qu'il a pu observer. Les informations sont transmises par le destinataire au *Router* sous la forme de messages de contrôle. Elles peuvent être demandées par le *Router* sous la forme de messages de commandes.

Un des problèmes de ce type est que le *Router*, sur base des informations qu'il détient, pourrait avoir la possibilité d'envoyer un message vers plusieurs destinataires.

[**Hohpe**] propose alors plusieurs solutions à mettre en oeuvre sous la forme de stratégies :

1. soit ignorer le fait et envoyer le message au premier destinataire. Cela comporte le risque d'envoyer tous les messages à un seul routeur ;
2. soit lister les différents destinataires susceptibles de recevoir le message en satisfaisant l'algorithme et en choisir un au hasard dans cette liste ;
3. soit envoyer le message à tous les destinataires pour lequels la règle est satisfaite.

Enfin, [**Hohpe**] n'évoque pas le cas où le message ne satisferait aucune règle. Nous allons donc poser que dans ce cas, le router dirigera le message vers un *Invalid Message Channel* que nous avons déjà décrit dans ce travail dans le chapitre relatif aux *Channel*s.

7.3.2 Paramètres qui peuvent influencer le routage d'un message

Nous pouvons raisonnablement poser que les paramètres suivants donnent une image de l'activité d'un composant :

- nombre de messages reçus (origine : *Port*) ;
- nombre de messages traités (origine : *MessageProcessor*) ;
- temps de traitement des messages (origine : *MessageProcesse* ;
- nombre de messages envoyés (origine : *Sender*) ;
- nombre de messages en attente dans *IncomingMessages* ;
- nombre maximum de messages dans *IncomingMessages* ;
- nombre de messages en attente dans *OutgoingMessages* ;
- nombre maximum de messages dans *OutgoingMessages* ;
- nombre de destinataires (origine : liste des destinations) ;
- date de réception du dernier message reçu (origine : *Port*) ;
- date de début de traitement du dernier message (origine : *MessageProcessor*) ;
- date de début de traitement du dernier message (origine : *MessageProcessor*) ;
- date de d'envoi du dernier message transmis (origine : *Sender*) et
- date et heure du début de collecte des informations.

Nous plaçons toutes ces informations dans une classe sérialisable : *RunningInformations*. Ces informations sont envoyées par un composant au *Router* sous la forme d'un message. Toutes ces informations, stockées dans le *Router* sont utilisées pour sélectionner le destinataire.

Périodiquement, le *Router* demandera ces informations via l'envoi d'un message de commande à chacun de ses destinataires. Le message envoyé sera doté d'un *ReturnAddressHeader* pour indiquer aux destinataires la destination de leurs messages de réponse. Cet intervalle de temps est stocké au sein du *Router* en tant qu'attribut.

Comme nous n'avons pas précisé le type du message et que ce message n'est pas présenté par [**Hohpe**], nous introduisons un nouveau type de message : le *ResponseMessageType*. Le *ResponseMessage* a le même format que le *CommandMessage* : un nom et une liste de paramètres.

7.3.3 Modélisation

Le *DynamicContentRouter* est une spécialisation du *Router*. Comme dans le *Content-Based Router*, le nombre de destinataires gérés est fonction de l'algorithme implémenté. Lorsque ce nombre ne correspond pas aux destinations configurées, le *Router* est dans l'état Invalid. L'attribut *timeToRequestInformation* est utilisé pour définir la fréquence de demande d'in-

formations.

La stratégie de choix est modélisée par l'attribut *destinationsStrategy*. Cet attribut possède trois valeurs :

1. envoi vers le premier destinataire trouvé ;
2. envoi vers l'un des destinataires possibles ;
3. envoi vers tous les destinataires qui satisfont les conditions.

7.4 Recipient List

7.4.1 Présentation

Ce type de router, représenté par le pictogramme de la figure 7.3, est chargé dès la réception d'un message de le retransmettre vers une liste de destinataires. Contrairement à l'idée initiale, le *Router* envoie cette fois un message à plusieurs destinataires.

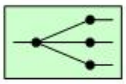

FIG. 7.3 – Pictogramme : RecipientList

Le créateur du message doit avoir une connaissance des destinataires possibles d'un message sous la forme de leurs identifiants au sein du routeur. Les destinataires sont identifiés par leur position au sein de la liste définie dans le DSL. Cela implique que le créateur du message connaisse le nombre de destinataires que possède le *Router* auquel il envoie son message. Cette information peut être obtenue à la demande selon le mécanisme utilisé pour le *Dynamic Router*.

Enfin, le message à traiter par ce router doit contenir un en-tête spécifique pour permettre ce type de routage.

7.4.2 Modélisation

Nous devons introduire un nouveau type de *Header* : le *RecipientListHeader* qui va contenir la liste des destinataires du messages. Le *RecipientListRouter* analyse le message reçu et envoie le message vers les destinataires demandés.

7.5 Splitter

7.5.1 Présentation

Certains messages comme dans notre exemple du constat d'excès de vitesse peuvent contenir des informations agglomérées. Ces informations, qui forment un tout, ne sont pas toujours destinées à un seul destinataire. L'agglomérat peut parfois contenir plusieurs informations de même type, placées dans une liste, pour lesquelles, il faudrait réaliser un même traitement distinct pour chaque élément de l'ensemble. Le *Splitter*, identifié par le pictogramme de la figure 7.4, est chargé de réaliser ce découpage.

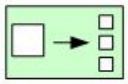

FIG. 7.4 – Pictogramme : Splitter

Ce router est chargé de séparer les différents contenus d'un message et de les envoyer vers les destinataires ad hoc. Il peut également être chargé d'ajouter un identifiant à tous les composants du message initial afin de pouvoir les relier dans un traitement ultérieur.

7.5.2 Modélisation

L'implémentation du *SplitterMessageProcessor* contient l'implémentation de l'algorithme de découpage du message. Les messages à envoyer utilisent le mécanisme standard. Encore une fois, le nombre requis de destinations doit être configuré afin que le composant soit en état de recevoir et traiter les messages.

7.6 Aggregator

7.6.1 Présentation

Dans le chapitre précédent, nous avons évoqué le *Splitter* de messages. Il nous faut maintenant évoquer l'*Aggregator*, dont le pictogramme est présenté à la figure 7.5 et qui a pour rôle l'agrégation de plusieurs messages en un

seul.

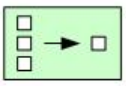

FIG. 7.5 – Pictogramme : Aggregator

L'agrégateur reçoit les messages à agréger et compose un seul message-résultat. Contrairement à la plupart des routeurs envisagés jusqu'ici, ce composant est dit *statefull* car il doit analyser tous les messages reçus avant de produire un résultat. L'agrégation se fait sur base d'un corrélateur (attribut de message que nous avons vu dans le cadre de ce travail).

Remarque : nous avons modélisé tous les routeurs de façon à ce qu'ils puissent être "statefull". Cela se produit lorsque *maxIncomingMessages* est plus grand que "1".

L'un des problèmes du traitement réalisé par ce composant est de savoir quand l'agrégateur peut effectuer son agrégation c'est-à-dire quand il a reçu toutes les parties du message à produire ou quand il peut considérer qu'il a reçu tout ce qu'il devait recevoir pour réaliser cet assemblage.

Selon [**Hohpe**], nous pouvons envisager plusieurs stratégies :
- attente de tous les messages : lorsque les messages contiennent outre le corrélateur, un indicateur du nombre de messages à attendre (par exemple, le nombre total de lignes à traiter est transmis avec chaque ligne) ;
- un timer : agrégation des messages reçus après un certain temps de non-arrivée de nouveaux messages ;
- "Premier meilleur" : lorsqu'il est possible d'évaluer la qualité des messages reçus, agrégation dès la réception du message considéré comme le meilleur message sans tenir compte d'autres messages pouvant arriver plus tard ;
- timer et meilleur message : attente d'un timer ou d'un message d'une certaine "valeur" (cas qui combine les deux cas précédents) ;
- arrivée d'un élément extérieur comme par exemple le changement de date dans le cas de messages liés aux événements d'un jour précis.

Toutes ces stratégies sont destinées à déterminer la qualité du message final. Cela se fait en tenant compte du fait que le réseau peut être indisponible et donc empêcher la livraison de certains de ces messages ou que certains composants peuvent ne pas répondre pour toute une série de raisons techniques ou fonctionnelles.

7.6.2 Modélisation

Ce type de *Router* est modélisé sous la forme de la classe *AggregatorRouter*, spécialisation de la classe *Router*. La recherche des messages pouvant constituer un aggrégat est confiée à une implémentation de l'interface *IAggregatorStrategy*. Pour pouvoir être valide, un tel agrégateur devra disposer dans sa configuration d'une implémentation concrète de cette interface.

7.6.3 Contrainte

le *Point-to-Point channel* doit avoir un seul port d'entrée et un seul destinataire :

```
Context PointToPointChannel
     inv: destinations->size() = 1
```

7.6.4 Exemple

Dans notre exemple de l'excès de vitesse, nous allons devoir agréger les données de l'infraction à celles du conducteur retrouvées sur base du numéro de plaque.

7.7 Resequencer

7.7.1 Présentation

Dans les routeurs que nous avons vu, nous n'avons rien dit à propos de l'ordre d'envoi des messages vers les destinataires. Nous avons toujours supposé que le message était traité et envoyé dès que le router le reçoit. Ainsi, il peut arriver, suite au traitement asynchrone, que les messages d'une séquence soient expédiés en désordre. Le but du *Resequencer*, représenté par le pictogramme de la figure 7.6, est de stocker les messages temporairement et de les réordonner avant envoi vers le destinataire.

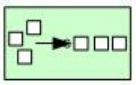

FIG. 7.6 – Pictogramme : Resequencer

Ceci implique que ce *Router* soit également "statefull" afin de pouvoir stocker les messages arrivés avant le message ayant le bon numéro au sein de

la séquence.

7.7.2 Modélisation

Nous modélisons cet élément comme une spécialisation du *Router*. un traitement interne sera nécessaire pour envoyer les messages dans l'ordre. Comme il envoie tous ces messages vers un seul destinataire, il aurait pu être placé dans la catégorie des *Channel*s.

7.7.3 Contraintes

- le *Point-to-Point channel* doit avoir un seul port d'entrée et un seul destinataire :
  ```
  Context PointToPointChannel
       inv: destinations->size() = 1
  ```
- le destinataire ne peut référencer le port d'entrée pour transmettre les messages (pour éviter le bouclage) ;

7.8 Composed Message Processor

7.8.1 Présentation

Le *Composed Message Router*, représenté par le pictogramme de la figure 7.7, a pour but de traiter un message composé de plusieurs parties, chaque partie nécessitant un traitement personnalisé et différent des autres.

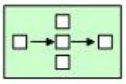

FIG. 7.7 – Pictogramme : Composed Message Processor

Ce *Router* reçoit ce type de messages, gérent les traitements de chaque partie puis compose le message résultat du traitement avant de l'envoyer vers son destinataire.

7.8.2 Modélisation

Comme ce processus pourra être implémenté de façon spécifique, nous n'en proposerons pas de modélisation. Nous laissons à l'utilisateur le soin de définir les ports d'entrée d'un tel processus et les composants internes

dont il a besoin pour transformer un message en un message composé : un *Splitter* semble nécessaire pour découper le message en ses parties, un *Content-based Router* pourrait être utile pour le routage vers les composants ad hoc et enfin un *Aggregator* devait servir à recomposer le message à partir des parties traitées.

7.9 Scatter-Gather

7.9.1 Présentation

Cette fois encore [**Hohpe**] nous introduit à un autre exemple de processus mettant en oeuvre une composition de routeurs. Toutefois, ce composant est orienté non plus vers le traitement de messages composés mais vers une fonction de récolte d'informations. Le *Scatter-Gather* va diffuser une demande d'information à plusieurs composants, puis dès réception des réponses de ceux-ci, il va utiliser un agrégateur pour présenter parmi les réponses reçues la meilleure offre.

7.9.2 Modélisation

Etant donné que le traitement effectif des messages se fera à l'aide de plusieurs composants, nous ne proposerons pas de modélisation spécifique à ce routeur.

7.10 Routing Slip

7.10.1 Présentation

Nous allons découvrir un routeur dont le rôle est le traitement d'un message par applications successives de traitements élémentaires. Le *Routing Slip*, représenté par le pictogramme de la figure 7.8, est chargé valider un message. La validation complète de ce message se fait en validant successivement certains éléments du message. Le message complet parcourt dès lors une suite de composants validants ou est rejeté par l'un d'eux.

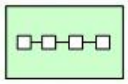

FIG. 7.8 – Pictogramme : Routing Slip

La méthode de validation se fait selon la séquence suivante :

1. validation de message par un composant A
2. soumission du message au validateur suivant (B) sauf si le message n'est pas valide pour le validateur A.

La validation complète du message suppose un point central de contrôle qui ne peut se faire qu'avec l'aide d'un *Content-Based Router*.

7.10.2 Modélisation

Comme pour les routeurs précédents, la modélisation de ce routeur se fait sous la forme d'un processus avec port d'entrée, d'un port de sortie. Chacun des composants accepte le message et le transmet au suivant en passant par un *Content-Based Router* central au processus ou refuse celui-ci et le transmet à un *Channel* chargé de traiter les messages invalides.

7.11 Process Manager

7.11.1 Présentation

Dans la série des routeurs dont le résultat est le fruit d'une combinaison de composants, voici le *Process Manager*. Le but de ce routeur est de présenter un traitement complet d'un message sous la forme d'un routage : chaque message reçu conduit à la transmission d'un message vers un destinataire.

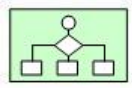

FIG. 7.9 – Pictogramme : Process Manager

Le *Process Manager* réalise le processus en orchestrant le traitement complexe du message à l'aide d'un ensemble de composants. Chacun des ces composants internes réalise une étape de l'ensemble. En fonction du résultat du traitement, le *Process Manager* sélectionne le prochain composant comme le faisant le *Routing slip* pour la validation par applications successives de valideurs.

7.11.2 Modélisation

A ce stade de notre travail, nous modéliserons ce composant comme un processus.

7.12 Message Broker

7.12.1 Présentation

[**Hohpe**] insiste sur le fait que les routeurs servent à découpler le composant émetteur du message à router du composant destinataire du message. Avec le *Message Broker*, [**Hohpe**] n'introduit pas de nouveau routeur mais insiste surtout sur le fait qu'un message est traité éventuellement avec appel(s) à d'autre(s) composant(s) mais où le routeur est le seul à maintenir un contrôle central sur l'ensemble des échanges de messages.

7.12.2 Modélisation

Nous ne modélisons pas ce pattern sous la forme d'un composant. Si nous avions besoin d'un tel composant, nous le modéliserions sous la forme d'un processus.

7.13 Remarques à propos des *Router*s

7.13.1 Réutilisation

Dans ce chapitre, nous avons vu quelques routeurs simples qui acheminent des messages sur base de leur contenu ou de conditions qui dépendent de l'état du routeur c'est-à-dire d'informations reçues d'autres composants. Nous avons également vu plusieurs processus, chargés de router les messages, dont le résultat est le fruit de plusieurs actions. Ceci valide l'introduction de la notion de processus que nous avions énoncée un peu plus tôt dans cette étude.

Toutefois, nous pouvons remarquer de la modélisation des routeurs que seul le routage réalisé au sein des routeurs simples, sous la forme d'implémentation d'un *MessageProcessor* est réutilisable. Le routage à l'aide de processus est lui généralement spécifique à ses éléments constitutifs. La seule méthode pour réutiliser des processus consisterait à leur ajouter des propriétés qui permettraient de les particulariser et par la même de les réutiliser comme nous le faisons avec les routeurs simples.

Nous pouvons également voir que la séparation proposée par [**Hohpe**], au sein de notre système de messagerie, s'estompe peu à peu : un *Message Channel* qui a initialement pour rôle la transmission d'un message d'un composant à un(plusieurs) autre(s) finit par avoir un rôle très proche de la fonction de routeur surtout quand nous dotons le routeur d'une possibilité de stockage de messages avant la sélection du destinataire.

7.13.2 Nombre de destinations nécessaires

Nous avons vu que plusieurs *Router*s avaient besoin d'un nombre bien précis de destinations. Ce nombre étant spécifique à l'implémentation du *Router*, nous imposons que chaque *Router* implémente l'interface *IRequired-Destinations*. Cette méthode, spécifique à chaque implémentation, retourne le nombre de destinations qui lui sont nécessaires. Nous pouvons généraliser l'usage de cette interface en posant que lorsqu'aucun contrôle du nombres de destinations n'est requis, la méthode renvoie une valeur convenue comme par exemple **(-1)**.

Quant à l'utilisateur des composants au niveau du DSL, un "simple" regard dans le code à défaut de documentation, le renseignera quant aux nombres de destinations à configurer.

Il est important de noter au sujet du DSL que cette interface ne fera pas partie des descriptions de la solution.

Chapitre 8

Transformation de messages

8.1 Introduction

Dans les parties précédentes de ce travail, nous avons vu comment acheminer les messages à travers le système. Nous nous sommes penchés sur les canaux de communication, la construction de messages et leur routage. Nous avons évoqué brièvement la transformation des messages dans la première partie de cette étude, transformation qui est un élément important pour permettre l'intégration de systèmes hétérogènes. Nous allons maintenant nous pencher sur cet aspect du système de messagerie.

8.2 Enveloppe Wrapper

8.2.1 Présentation

Revenons à la création du message par une application : lorsque cette application veut communiquer avec une autre application, nous avons posé que l'application émettrice créait un message et le transmettait à un destinataire via un *EndPoint*. Le message est ensuite envoyé à travers le système de messagerie afin d'être acheminé de façon transparente vers l'application destinataire de ce message.

A ce stade de l'exposé, nous pouvons poser que le message envoyé est compréhensible par les deux applications. Toutefois, ce message est composé de headers propres au dialogue entre les deux applications. Nous avons également vu que certains éléments du *Header* étaient également utilisés pour acheminer le message à travers le système de messagerie. Nous pourrions dès lors avoir dans le message original deux types de *Header*s.

Afin d'éviter d'altérer le message à transmettre (ce qui implique vu notre postulat la création de multiples messages) et de devoir gérer plusieurs "types" de header, [**Hohpe**] introduit le concept d'*Envelope Wrapper* représenté par le pictogramme de la figure 8.1.

FIG. 8.1 – Pictogramme : Envelope Wrapper

Le message original, c'est-à-dire *Body* et *Header*s, est encapsulé dans un nouveau message dont il constitue le *Body*. L'encapsulation peut prendre la forme d'un stream ZIP par exemple. Ainsi le nouveau message peut avoir ses propres headers utilisés pour l'acheminement du message.

En utilisant ce principe, le message original pourra être enrobé par plusieurs enveloppes selon les besoins pour le transport de ce message. Une fois arrivé à destination, le *Body* sera extrait du message (un composant ajoute ou retire une seule enveloppe à chaque traitement) et transmis à l'application ou au composant destinataire.

Nous devrons donc utiliser un *Envelope Wrapper* pour encapsuler le message avant la transmission à travers le système de messagerie et en fin de transfert un deuxième *Envelope Wrapper* pour extraire le message de son enveloppe.

8.2.2 Modélisation

L'*Envelope Wrapper* est modélisé comme une spécialisation de la classe *Translator*. Il devra être accompagné d'une implémentation de l'interface *IEnveloppeWrapper* qui prendra en charge la transformation d'un message.

8.3 Content Enricher

8.3.1 Présentation

Après avoir traité le problème du transport du message, nous nous intéressons au problème d'enrichissement d'un message. Certains messages émis par une application ne peuvent être compris par l'application réceptrice à moins d'y ajouter des informations supplémentaires. Ceci est le cas par exemple quand on utilise une information relative à un pays dans une adresse : l'application émettrice pourrait connaître seulement le numéro de

ce pays ou de la région tandis que l'application réceptrice pourrait avoir besoin du code de cet Etat.

Plusieurs solutions sont possibles pour résoudre ce problème :
- soit de fournir à l'application émettrice l'information manquante. Ceci a pour conséquence la réplication de cette information et la modification de l'application émettrice du message ;
- soit d'établir un couplage entre l'application émettrice et l'application qui détient l'information à ajouter ;
- soit, enfin, de transmettre dans un premier temps le message à l'application qui connaît le lien entre le numéro de l'état et son code puis dans un second temps de transmettre le message complété à l'application destinataire.

Le rôle d'enrichir un message est confié au *Content Enricher*, représenté par le pictogramme de la figure 8.2.

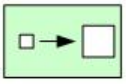

FIG. 8.2 – Pictogramme : Content Enricher

Ce composant enrichit l'information contenue dans le message en utilisant une ressource extérieure avant de le transmettre à son destinataire. Cette ressource pourra être une base de données avec laquelle il dialogue à l'aide de messages ou une autre application qui gère par exemple des données de référence.

Le *Content enricher* reçoit le message à traiter, obtient l'information à y ajouter et transmet le tout à un seul destinataire.

8.3.2 Modélisation

Comme les possibilités de sources de données sont trop nombreuses pour être modélisées en une seule classe, nous décidons que ce composant sera modélisé sous la forme d'un processus. Nous laissons à l'utilisateur du système, le rôle de composer le processus selon ses besoins.

8.4 Content Filter

8.4.1 Présentation

Après le composant qui enrichit, nous analysons le *Content Filter* représenté par le pictogramme de la figure 8.3.

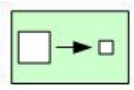

FIG. 8.3 – Pictogramme : Content Filter

Ce composant crée un message à partir d'informations lues dans le message reçu. Seules certaines parties du message initial sont transmises à l'application destinataire. Cela peut être utile lorsque le message contient beaucoup d'informations dont une grande partie est inutile à l'application destinataire. Son rôle est de simplifer le message. Ceci peut être utile pour réduire les données transportées et pour éviter de diffuser certains informations sensibles à des applications qui n'en ont pas l'usage.

8.4.2 Modélisation

Ce composant est une spécialisation du *Translator*. Il utilise une implémentation de l'interface *IContentFilter* pour réaliser le retrait des parties de messages.

8.5 Claim Check

8.5.1 Présentation

Dans le même but de simplifier le message transporté, voici le *Claim Check* représenté par le pictogramme de la figure 8.4.

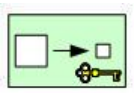

FIG. 8.4 – Pictogramme : Claim Check

Ce transformateur reçoit un message et le transforme afin de le limiter à quelques informations. Le nouveau message est alors transmis à un des-

tinataire. Ce destinataire va alors renvoyer un message de type "réponse" vers le *Claim Check* qui recomposera la réponse sur base des informations du message initial qu'il avait mémorisé entretemps. Le lien entre la demande et la réponse se faisant à l'aide d'un identifiant placé par le *Claim Check*. La simplification du message peut aussi être utile quand on ne veut pas transmettre à une application des informations confidentielles contenues dans le message original.

8.5.2 Modélisation

Etant donné que le pattern *Claim Check* est implémenté par un processus étant donné qu'il utilise notamment un *Content Filter* ou un *Splitter* pour envoyer le message simplifié et un *Aggregator* pour recomposer le message initial et la réponse.

8.6 Normalizer

8.6.1 Présentation

Enfin, pour pouvoir traiter des informations venant de plusieurs sources, nous pouvons utiliser un composant qui normalise les messages. Toutefois, comme il ne semble pas possible de transformer tous les types de messages en utilisant un seul composant, nous allons poser que le *Normalizer*, représenté par le pictogramme de la figure 8.5, transforme chaque message en un message de type standard pour notre système.

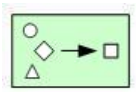

FIG. 8.5 – Pictogramme : Normalizer

8.6.2 Modélisation

Nous modélisons ce composant comme une spécialisation du *Translator*. Pour pouvoir fonctionner, il va s'appuyer sur une implémentation de l'interface *INormalizer* qui réalise la transformation.

Si nous devions modéliser un ensemble de types de messages, nous pourrions utiliser un processus. Ce processus serait composé d'un *ContentBasedRouter* qui orienterait les messages selon leur type ou format vers l'un des *Normalizer* du processus.

8.6.3 Exemple

Nous pourrions devoir normaliser un message contenant du texte libre en format XML.

8.7 Canonical Data Model

[**Hohpe**] présente ensuite un concept : le *Canonical Data Model*. Il s'agit de mettre en place à la sortie des ports d'émission des applications impliquées dans un système, une(des) transformation(s) vers un modèle canonique de messages c'est-à-dire un format de message normalisé pour le système. Ces messages, une fois standardisés peuvent être plus facilement transportés à travers le système.

Chapitre 9

Endpoints

9.1 Introduction

Après avoir vu comment les messages étaient acheminés et traités au sein de notre système, voyons en détail les composants que nous propose [**Hohpe**] pour permettre la communication de ce système avec les applications qui l'utilisent. Pour rappel, le rôle d'un *EndPoint* est de permettre la communication d'une application normalement déjà existante avec le système de messagerie, système qui joue le rôle d'intermédiaire entre deux applications.

9.2 Messaging Gateway

9.2.1 Présentation

Le *Messaging Gateway* est le *Endpoint* type : il permet l'échange d'informations entre l'application qui utilise le système et le système proprement dit.

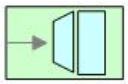

FIG. 9.1 – Pictogramme : Messaging Gateway

9.2.2 Modélisation

Le *MessageGateway* est une spécialisation du *Endpoint*. Il s'appuiera sur une couche qui obtient les informations de l'application avec laquelle il dialogue. Cette couche est matérialisée par une implémentation de l'interface

IApplicationInterface.

9.3 Messaging Mapper

9.3.1 Présentation

Le type de *Endpoint* suivant est le *Message Mapper*. Il s'agit d'un composant chargé de la mise en correspondance de l'information transmise par l'application, généralement sous la forme d'objets propres au domaine de l'application, et le message à envoyer vers le système de messagerie.

9.3.2 Modélisation

Comme nous avons prévu cette mise en correspondance dans la couche d'interface qui joue le rôle de port pour un *Endpoint* client ou d'émetteur de message pour un *Endpoint* fournisseur, nous ne modélisons pas en tant que tel ce type de *Endpoint*.

9.4 Channel Adapter

9.4.1 Présentation

Nous retrouvons maintenant le *Channel Adapter* déjà proposé par [**Hohpe**]. Au vu de la définition du *Endpoint* sous la forme du *Messaging Gateway*, nous voyons qu'il s'agit en fait de la combinaison de deux *Endpoint*s qui dialoguent entre eux. Nous pouvons dès lors raisonnablement poser que l'information échangée entre eux aura une forme bien définie et que deux *Endpoint* pourront dialoguer entre eux quand ils appartiennent chacun à un système de messagerie différent.

9.4.2 Modélisation

Comme il s'agit de réutiliser des composants déjà définis, nous ne proposerons pas de modélisation spécifique à ce type de *Router*.

9.5 Transactional Client

9.5.1 Présentation

Ce type de *Endpoint* prend en charge la transaction entre le système de messagerie et l'application client.

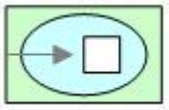

FIG. 9.2 – Pictogramme : Transactional Client

La transaction englobe l'obtention d'informations jusqu'à la délivrance du message au système ou de la réception du message à la fourniture à l'application des informations.

Le dialogue entre l'application, un *EndPoint* et le système peut être

représenté par le diagramme de séquence de la figure 9.3.

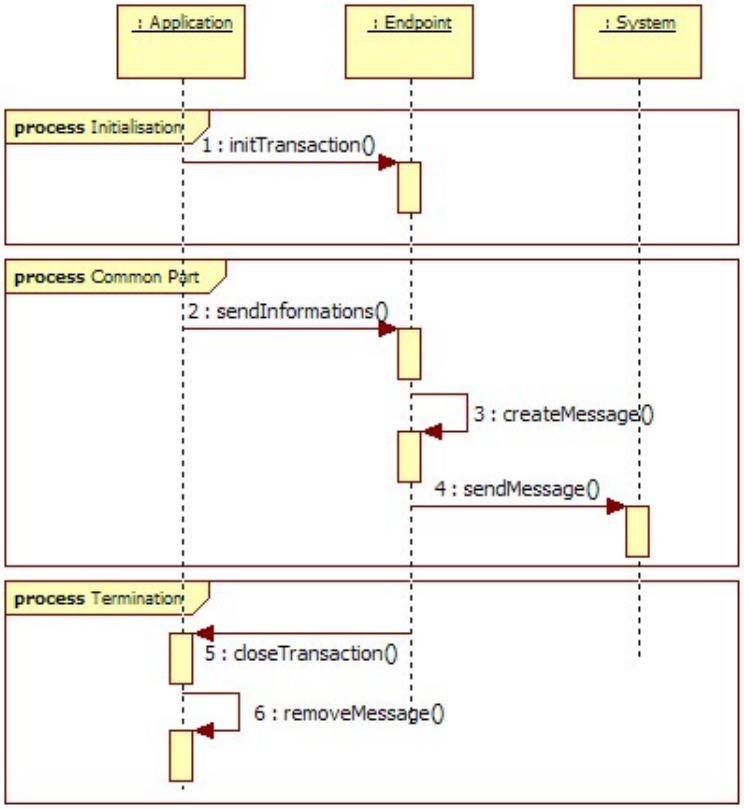

FIG. 9.3 – Diagramme de séquence : Dialogue entre constituants d'un *End-Point*

9.5.2 Modélisation

Nous ne modélisons pas ce *Endpoint* comme un composant indépendant mais comme une option sous la forme de l'attribut *requiresTransaction* des *Endpoints* : lorsque nous déciderons d'activer cette option, le *Endpoint* placera l'échange dans une transformation. Il n'y aura pas de transaction quand cette option ne sera pas activée.

9.6 Polling Consumer

9.6.1 Présentation

Comme nous avons déjà introduit cette notion lors de notre présentation, nous ne modéliserons pas ce pattern, représenté par le pictogramme de la figure 9.4.

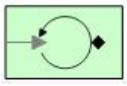

FIG. 9.4 – Pictogramme : Polling Consumer

9.7 Event-Driven Consumer

9.7.1 Présentation

Tout comme le pattern du *Polling Consumer*, nous avons déjà introduit ce pattern, représenté par le pictogramme de la figure 9.5 lors de l'introduction du *Endpoint* au chapitre 4.

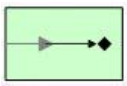

FIG. 9.5 – Pictogramme : Event-Driven Consumer

9.8 Competing Consumers

9.8.1 Présentation

En nous présentant ce *Router*, [**Hohpe**] propose une solution à l'engorgement d'un composant en distribuant le traitement à plusieurs composants équivalents. Il propose plusieurs solutions de distribution de messages. Le *Message dispatcher* décrit ci-après en est une.

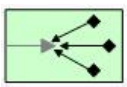

FIG. 9.6 – Pictogramme : Competing Consumers

Ce type de *Endpoint* n'est pas vraiment utile à notre système car nous avons prévu qu'un tel processus de distribution pouvait être réalisé par un *Router*, composant interne, qui peut distribuer les messages de façon dynamique et notamment en fonction d'une charge sur ses destinataires possibles. Il n'est donc pas modélisé en tant que composant.

9.9 Message Dispatcher

9.9.1 Présentation

Comme nous n'utiliserons pas a priori le *Competing Consumer*, il paraît logique de ne pas s'attarder sur ce type de *EndPoint* qui implémente justement un des *Competing Consumers* invoqués dans ce point.

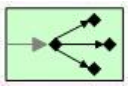

FIG. 9.7 – Pictogramme : Message Dispatcher

9.10 Selective Consumer

9.10.1 Présentation

Le *Selective Consumer* est en fait un *EndPoint* qui filtre les messages avant de transmettre l'information qu'ils contiennent à l'application destinatrice. Comme nous pouvons implémenter des processus et que nous disposons de composants qui filtrent les messages, nous n'implémenterons pas ce composant de façon spécifique mais nous définirons plutôt un processus pour réaliser cette tâche.

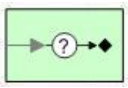

FIG. 9.8 – Pictogramme : Selective Consumer

9.11 Durable Subscriber

9.11.1 Présentation

Un *Durabler Subscriber* est un composant qui stocke les messages jusqu'à ce qu'il soient délivrés à un destinataire qui peut ne pas être connecté. Ceci permet d'accroître la robustesse du système.

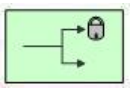

FIG. 9.9 – Pictogramme : Durable Subscriber

9.11.2 Modélisation

La modélisation de ce composant est assez évidente : nous fournissons une implémentation de l'interface *IMessageStorage*. Cette interface décrit les méthodes responsables de la sauvegarde des messages de façon persistante. Aucune nouvelle modélisation n'est dans ce cas nécessaire.

9.12 Idempotent Receiver

9.12.1 Présentation

Avec l'*IdemPotent Receiver*, [**Hohpe**] évoque le cas d'un message qui arrive deux fois chez un même destinataire. Ceci peut arriver lors de reprise sur panne ou encore lorsque un message a bien été reçu par le destinataire mais que l'accusé de réception n'est pas parvenu à l'émetteur qui renvoie de ce fait le message.

Nous rencontrons à nouveau le problème de l'élimination d'un même message reçu en plusieurs exemplaires. [**Hohpe**] nous explique que devoir éliminer un message est coûteux en terme de ressources car il faut enregistrer le message et filtrer ceux-ci sur base de ceux qui ont déjà été autorisés. Ce filtrage implique que le système de stockage des messages, pour éviter qu'il ne soit trop gros, soit limité dans le temps d'où une option qui fixe le temps de garde des messages transmis.

[**Hohpe**] propose pour arriver à cette idempotence de définir des messages de façon telle qu'une double réception ne cause pas de problème lors du traitement. En reprenant son exemple, plutôt que d'envoyer un message "Retirer 100 euros du compte Dupont"', il faut envoyer un message "Placer le solde du compte Dupont à 130 euros".

9.12.2 Modélisation

Comme nous ne connaissons pas à ce stade, la forme des messages, nous nous limitons dans le cadre de ce travail à détecter les messages en double et à les éliminer. Or, dans le chapitre des *Router*s, nous avons déjà modélisé un filter. De ce fait, nous n'implémentons pas ce type de *Endpoint* en tant que tel et nous le remplaçons par une combinaison d'un *Endpoint* classique suivi d'un filtre.

9.13 Service Activator

9.13.1 Présentation

Dans certains cas d'intégration, une application peut faire appel à plusieurs services. Afin de devoir effectuer différents types d'appels aux *Endpoint*s du système de messagerie, [**Hohpe**] propose de développer un *Service Activator*.

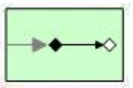

FIG. 9.10 – Pictogramme : Service Activator

Cet *Endpoint* masque à l'application les différents types de service en un seul service qui agit à la façon du design pattern "Facade" [**GOF**] : à un seul point d'entrée, correspondent plusieurs services ; chacun d'eux est connu du *Endpoint* qui fait le routage et la mise en forme des messages.

[**Hohpe**] signale que ce composant peut agir de deux façons distinctes :
- de façon synchrone (bloquante) en transmettant la requête au système de messagerie et en bloquant le traitement de l'application jusqu'à ce qu'elle reçoive la réponse du système via un message provenant du système.
- de façon asynchrone en transmettant la requête au système et en fournissant la réponse à ce système dès réception de la réponse émise par le système.

9.13.2 Modélisation

Mode Synchrone

Utiliser le *Service Activator* pour fournir un service synchrone implique que l'application client puisse dialoguer de façon active avec le *Endpoint*. Comme ce dialogue repose sur la couche d'interface, il suffit que l'application qui souhaite dialoguer avec le *Endpoint* utilise une méthode construite à cet effet. La méthode après avoir reçu les informations va attendre que la réponse lui soit envoyée avant de rendre le contrôle à l'application appelante. Nous ne devons donc pas modéliser un tel type de *Endpoint*.

Mode Asynchrone

Lorsqu'une application dialogue en mode asynchrone, elle utilise en fait deux *Endpoint*s : un *Endpoint* client et un *EndPoint* fournisseur. Comme nous disposons déjà de ces composants, nous n'avons rien de neuf à modéliser.

Remarque : nous avons bien implémenté un *Service Activator* vu que le *Endpoint* implémente bien les deux types de service en un seul composant.

Chapitre 10

Gestion du système de messages

10.1 Introduction

Après les messages et les composants qui les créent, les acheminent et les transforment, voyons maintenant les composants utiles pour la gestion du système proprement dit c'est-à-dire le contrôle et la gestion, l'obtention d'informations pour la vérification statique et dynamique de leur bon fonctionnement et éventuellement pour réaliser des tests.

10.2 Control Bus

10.2.1 Présentation

Le *Control Bus* est le composant qui sert à monitorer le système et à modifier les paramètres des composants qui le composent. Cela implique que chacun des composants du système soit connecté d'un part au flot des messages et d'autre part au *Control Bus*.

FIG. 10.1 – Pictogramme : Control Bus

Ce composant pourra alors envoyer ou recevoir de chacun des composants du système des messages pour

- configurer un composant. Par exemple pour activer une option de persistance des messages dans un composant ;
- vérifier que le composant fonctionne en recevant de façon périodique des messages de type 'hello' par exemple ;
- introduire dans le système des messages de tests ;
- obtenir des informations sur les exceptions rencontrées par un composant ;
- obtenir des statistiques.

Enfin, n'oublions pas l'un des rôles les plus évidents, présenter un état du système aux personnes chargées de la maintenance du système.

Nous utilisons ce composant pour mettre de la "glue" entre tous les composants. Ce composant est la console du système comme décrit par [**Hohpe**] et nous permet d'installer dans le système les composants, de les arrêter et de suivre de façon régulière leur fonctionnement correct. Le *Control Bus* devient ainsi par extension *Le* système de messagerie.

10.2.2 Modélisation

Le *Control Bus* est modélisé à l'aide d'une classe indépendante des autres classes définies jusqu'à présent. Comme il est utilisé pour charger la description du système d'intégration, il ne fait pas partie du DSL.

10.2.3 Remarque

Nous avons parlé dans cette partie de messages de configuration. Comme ils ne font pas l'objet du propose de [**Hohpe**] ni de celui du DSL, nous n'approfondissons pas ce sujet mais le lecteur pourra trouver en annexe quelques exemples de tels messages.

10.3 Detour

10.3.1 Présentation

Ce composant est utilisé pour détourner les messages à certains moments notamment pour des fonctions de contrôle ou de test. Il sert à changer la route normale des messages pour les envoyer vers de nouveaux composants

utilisés pour la vérification des messages acheminés.

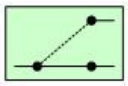

FIG. 10.2 – Pictogramme : Detour

10.3.2 Modélisation

Nous modélisons ce composant comme un *Dynamic Router* : dès la réception d'un signal venant de la console, le *Router* change la destination des messages qu'il doit retransmettre. Un autre signal permet la restauration du destinataire initial.

10.4 Wire Tap

10.4.1 Présentation

Le *Wire Tap* est un composant destiné à transmettre les messages au *Control Bus* sans altérer le flot normal des messages.

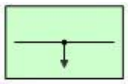

FIG. 10.3 – Pictogramme : Wire Tap

10.4.2 Modélisation

Pour envoyer un message vers deux destinations, nous pouvons déjà utiliser un *Publish-subscribe Channel*. Nous ne modélisons donc pas de nouveau composant pour effectuer cette tâche.

10.5 Message History

10.5.1 Présentation

Dans certains cas, et notamment pour faire un suivi de flot, il est nécessaire d'obtenir le chemin parcouru par un message. Ainsi [**Hohpe**] nous montre un exemple où le message est enrichi au passage de chaque composant du nom de ce composant.

10.5.2 Modélisation

Nous modélisons cette information sous la forme d'un *Header*. Ce *Header* a comme nom *History* et contient le nom du composant et le nom du port qui l'a reçu. L'ajout de ces informations au message peut être confié au port. Il s'agit d'une option qui lui est ajoutée.

10.6 MessageStore

10.6.1 Présentation

Après avoir suivi le chemin parcouru par un message, nous pouvons également souhaiter obtenir de l'information sur les messages envoyés par un composant et enregistrer ceux-ci dans une base de données par exemple.

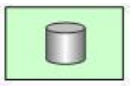

FIG. 10.4 – Pictogramme : Message Store

10.6.2 Modélisation

Nous modélisons un *MessageStore* comme un *InvalidMessage* ou un *DeadLetter*. Il a un port de réception des messages et aucune destination. Il est équipé d'une implémentation de l'interface *IGuaranteedDelivery* pour mémoriser les messages qu'il a reçu.

10.7 Smart Proxy

10.7.1 Présentation

Avec le *Smart Proxy*, [**Hohpe**] nous propose un composant qui permet de suivre la requête et la réponse quand la requête contient une adresse de retour(*Return Address*). Ceci peut poser problème car en utilisant le mécanisme d'adresse de retour, il est difficile d'intercepter la réponse.

10.7.2 Modélisation

Nous modélisons ce composant à l'aide d'un processus. Les réponses sont reçues par un composant interne qui remplace l'adresse de retour par

l'adresse d'un port du processus et place l'adresse de réponse initiale en mémoire. Ainsi, la réponse est envoyée vers ce port et plus vers l'adresse demandée. Ensuite une autre élément du *Smart Proxy* utilise le *Correlation Identifer* pour retrouver l'adresse de retour initiale et renvoie la réponse vers le bon destinataire. La requête et la réponse reliée peuvent alors faire l'objet du traitement de contrôle requis.

10.8 Test Message

10.8.1 Présentation

Avec le *Test Message*, [**Hohpe**] présente explicitement un pattern et plus simplement un composant implémentable. Il utilise le pictogramme de la figure 10.5 pour le représenter.

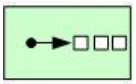

FIG. 10.5 – Pictogramme : Test Message

Ce pattern est utilisé pour tester le bon fonctionnement d'un composant. Il repose sur l'utilisation de plusieurs composants :

- un *Test Data Generator* qui crée les messages de tests à utiliser pour le test ;
- un *Test Message Injector* qui insère les messages de tests dans le système de messagerie ;
- un *Test Message Separator* qui sépare les messages de tests c'est-à-dire les réponses aux requêtes envoyées vers le composant à tester des messages réels traités par ce même composant ;
- un *Test Data Verifier* qui vérifie que les réponses reçues correspondent aux requêtes de test.

10.8.2 Modélisation

La modélisation d'un tel pattern se fait à l'aide de plusieurs composants :

- un composant, à développer, génère les messages de tests via un *Message Processor* et injecte les messages dans le système via un mécanisme standard. Avant l'injection, il place les messages de tests dans un système de données persistantes ;
- le *Test Message Separator* est en fait un *Router* placé dans le système et

– le *Test Data Verifier* est un composant également à développer. Il reçoit les réponses de tests, consulte le système de données où ont été placées les requêtes et effectue la vérification.

10.9 Channel Purger

10.9.1 Présentation

Lorsqu'il est nécessaire d'effectuer des tests dans un système, il apparaît certains messages non souhaités. Il peut s'agir de messages qui se trouvent encore dans les *Channel*s et qui perturbent les tests. Grâce au *Channel Purger*, représenté par le pictogramme de la figure 10.6, [**Hohpe**] propose de vider les channels à utiliser pour effectuer le test.

FIG. 10.6 – Pictogramme : Channel Purger

10.9.2 Modélisation

Dans notre modélisation des *Channel*s, nous avons prévu deux zones de stockage des messages : *IncompingMessages* et *OutgoingMessages*. Une vidange de ces messages implique de vider ces deux zones. Afin de réaliser la vidange de ces deux zones, nous demandons au *Control Bus* de générer un message qui demande ce traitement. Le message peut contenir un identifiant de la zone à vider : soit la zone *IncomingMessages*, soit la zone *OutgoingMessages* soit les deux zones. Nous pouvons également placer dans ce message une règle comme celle que nous avons utilisée pour le *Dynamic Content Router* afin de supprimer uniquement les messages répondant à une condition.

10.10 Synthèse sur les composants proposés

A part l'introduction du *Control Bus*, nous n'avons pas introduit dans ce chapitre de nouveaux composants mais plutôt des éléments à ajouter à ceux déjà existants.

Avant de nous plonger dans l'écriture du DSL relatif aux systèmes d'intégration, nous allons d'abord nous pencher sur quelques solutions permettant l'intégration d'applications et sur quelques notions relatives aux DSL et à la

modélisation. Ensuite, à l'aide du diagramme de classes reprenant tous les composants, nous définirons le DSL.

Chapitre 11

Solutions techniques d'intégration d'applications

11.1 Introduction

Comme le livre de [**Hohpe**] est principalement consacré aux patterns d'intégration en entreprise, nous allons d'abord passer en revue plusieurs solutions actuellement disponibles au sein des entreprises. Nous aborderons ensuite les points de vue sur les langages spécifiques aux domaines et aux outils de modélisation.

11.2 Web Services

11.2.1 Présentation

L'une des solutions techniques fréquemment utilisées pour l'intégration d'applications en entreprise est l'usage des services Web. Un service Web est un service mis à disposition par un serveur et utilisé par un client. Il est décrit formellement à l'aide du *Web Service Definition Language* ou [**WSDL**] dont les évolutions sont gérées par le [**W3C**]. Il permet l'échange de messages à travers l'Internet. Le fichier de définition, appelé aussi par extension *fichier WSDL*, décrit les méthodes offertes par le service, le type de protocole utilisé pour appeler ces dernières, le type de dialogue entre le client et le serveur (requête-réponse, one-way, notification, ...) et enfin fournit la définition des messages échangés pour chacune des fonctions proposées.

En général, le client communique avec le serveur à l'aide du protocole HTTP mais d'autres protocoles sont autorisés comme notamment JMS et SMTP. Les messages XML sont spécifiés en utilisant la norme [**XMLSchema**]

également suivie par le [**W3C**].

11.2.2 Mise en oeuvre

Il existe deux méthodes pour le développement d'un service Web :
- à partir du code développé en langage [**Java**], le développeur génère un fichier WSDL ;
- à partir du fichier WSLD décrit par l'analyste en collaboration avec l'architecte et les différents partenaires impliqués par ce service, le développeur génère les classes utiles pour les méthodes du services. Ensuite, il implémente la couche "business" pour la réalisation du service. On appelle cette approche *Contract First* car elle met en évidence la notion de contrat passé entre le fournisseur et le client du service.

L'utilisateur du service web utilise le WSDL pour la définition des classes (stubs) utile pour appeler le service. Il doit ensuite utiliser ces classes pour réaliser l'appel du service en utilisant l'URL du serveur où est déployé le service.

Des outils dans la plupart des langages actuels tels que Java, Microsoft .net ou encore PHP et Ruby permettent la mise en oeuvre pratique de services Web.

11.2.3 Avantages et inconvénients

La mise en place de services Web et leur utilisation impliquent une modification des applications existantes qui ne communiquent pas entre elles. Cela implique le développement spécifique de solutions dédiées à une communication. Cette solution peut, lorsqu'il s'agit d'intégrer d'anciennes applications, s'avérer complexe à mettre en oeuvre.

Par contre, l'utilisation de services avec un contrat clairement établi permet de lever toute ambiguité entre les partenaires. De plus, l'usage de ce type de technologie renforce la standardisation des développements au sein d'une entreprise.

11.3 Service Oriented Architecture

11.3.1 Présentation

Après les services web, nous pouvons aborder SOA. Il ne s'agit pas à proprement parler de technologie directement implémentable mais d'une focalisation méthodologique sur la distribution de services et leur niveau de

granularité. Comme nous avons plutôt orienté ce travail sur l'aspect pratique, nous ne nous pencherons pas plus en profondeur sur ce point.

Notons toutefois que l'un des principes de SOA est de construire des services à grande valeur ajoutée à partir de services de moins grande valeur jusqu'à se baser sur des services élémentaires comme la recherche d'informations dans une base de données. Tous les services, de base ou à haute valeur ajoutée, doivent rester indépendants c'est-à-dire reliés par un couplage le plus faible possible. Cette exigence implique de les faire communiquer à l'aide de technologie de type *services web*.

11.4 Enterprise Service Bus

11.4.1 Présentation

Une autre forme d'intégration d'application consiste à utiliser un bus pour réaliser l'intégration des applications. Un bus peut être vu comme une couche d'intermédiation qui offre un ensemble de services dont notamment différents connecteurs pour permettre la connection des systèmes, des services de routage, de transformations de messages.

Sur un bus, nous pouvons rencontrer les éléments décrits par les patterns proposés par [**Hohpe**]. Nous pouvons également y trouver les *Endpoint*s qui sont les connecteurs sur le bus des différents éléments.

11.4.2 Mise en oeuvre

Nous pouvons trouver de nombreuses implémentations de Service Bus sur le marché et notamment chez Oracle, BEA et Microsoft. Il existe également des solutions de type Open-sources telles que Mule (`http://mule.mulesource.org/display/MULE/Home`).

11.4.3 Avantages et inconvénients

Toutes ces solutions apportent à l'utilisateur, outre une interface graphique, de nombreux outils graphiques qui permettent un développement rapide et facilitent la mise en production de telles solutions. Elles apportent également des outils de monitoring comme ceux présentés au pattern *Control Bus*.

L'usage de ces solutions exige souvent une infrastructure relativement lourde avec par exemple des machines de développement équipées d'une mémoire de 4 gigabits.

Note : [**Hohpe**] mentionne le livre de [**ChappellESB**] comme prolongement de son livre.

11.5 Service Component Architecture

11.5.1 Présentation

Service Component Architecture (SCA) is a set of specifications which describe a model for building applications and systems using a Service-Oriented Architecture (SOA) selon la page du site qui coordonne les efforts d'un groupe d'industriels actifs dans le monde SOA (`http://www.oasis-opencsa.org/sca`).

Cette spécification utilise comme élément de base le composant. Un composant qui est une *instance d'implémentation* ([**SCA**]) est constitué de services (point d'entrée du message), de références (vers un point d'entrée d'un autre composant) et de propriétés. Plusieurs composants peuvent être assemblés pour obtenir un *Composite*. Un *Composite* peut lui-même être utilisé dans la définition d'un autre *Composite*. Enfin, [**SCA**] définit un *SCA domain* comme le résultat de l'assemblage de *composite*s de haut niveau. La communication entre composants(composites) propose plusieurs modes de dialogue : request-reply, one-way, ... tels que ceux décrits dans la norme WSDL.

11.5.2 Mise en oeuvre

La spécification définit, via les documents disponibles surle site, plusieurs implémentations de base notamment pour les composants écrits en langage Java ainsi qu'un ensemble d'éléments plus spécifiquement dédiés à l'assemblage des composants, aux transactions et aux connecteurs vers d'autres outils/normes tels que entre autre BPEL et WSDL. La spécification WSDL est d'ailleurs utilisée pour la définition des interfaces (services).

11.5.3 Avantages et inconvénients

La spécification est très ouverte aux différents standards (service Web et RMI par exemple). Elle est annoncée comme implémentable en différents langages de programmation.

Toutefois, elle est relativement récente ce qui pourrait amener dans un futur proche, des incompatibilités entre implémentations lorsque la spécification laisse une zone non entièrement spécifiée.

11.6 Apache Camel

11.6.1 Présentation

Apache Camel est une implémentation des patterns proposés par [**Hohpe**] ; il s'appuie sur des outils existants et notamment sur le framework Spring. Il propose plusieurs types de mapping pour la transformation de messages et utilise plusieurs protocoles pour le transport de messages tout en offrant des connecteurs de type service Web. En outre, Apache Camel propose un DSL exprimé en Java.

11.6.2 Mise en oeuvre

Le site propose plusieurs tutoriels afin de guider l'utilisateur. On retiendra cependant que l'utilisation d'une application consiste à créer un contexte, y définir une route pour les messages, route composée des composants proposés par [**Hohpe**] et d'exécuter le contexte pour activer la plate-forme.

11.6.3 Avantages et inconvénients

Solution open-source, elle offre l'avantage du coût de licence. De plus, son implémentation en Java la rend facilement accessible à de nombreux programmeurs. Enfin, elle supporte les langages de scripting comme Ruby ou PHP.

Toutefois, elle est moins supportée que SCA qui s'appuie sur un groupe d'industriels. Ceci pourrait la rendre plus fragile notamment au niveau évolution et support (nous trouvons seulement deux entreprises qui offrent du support) bien qu'elle s'appuie sur le framework Spring qui devient, dans le monde Java, un standard de facto.

Chapitre 12

Langage spécifique au domaine

12.1 Introduction

Souvent, en informatique, les applications sont développées sur base de besoins spécifiques et à l'aide de langages généralistes. Afin d'éviter le développement de toute nouvelle application à partir des constructions de base du langage, les développeurs ont peu à peu développé des librairies de code.

Ensuite, certains ont développé des méthodes de programmation pour aboutir finalement aux langages dits de quatrième génération ou encore programmation orientée objet. Dans ce type de langage, les développeurs manipulent des objets informatiques qui, par leur état et leur comportement, miment les objets de la vie réelle.

Toutefois, ces objets, concepts relativement pratiques pour les développeurs sont trop génériques et trop éloignés de leurs utilisateurs réels et des concepteurs d'applications. Ainsi est apparue le concept de "Langage spécifique au domaine d'application", ou "Domain Specific Language". Le principe étant de définir un langage spécifique au domaine concerné par les applications qui correspond directement aux concepts manipulés par les analystes et utilisateurs de ces applications.

12.2 Modèle

Le but premier de ce travail étant d'obtenir un langage spécialement dédié à notre système de messagerie en particulier et de façon plus générale à l'intégration d'applications, nous devons décrire les différents objets qui doivent faire partie de ce langage.

Dans un premier temps, nous avons établi une représentation de la situa-

tion que nous avons rencontrée avec notre exemple introductif. Nous appelons cette première représentation un *modèle* car chaque objet décrit est une représentation de l'objet tel que nous l'avons rencontré dans cette situation précise.

Ensuite, nous avons généralisé la description de chacun des éléments rencontrés. Nous avons ainsi produit une image ou encore un modèle de notre modèle initial. En langage de modélisation, ce modèle est appelé un metamodèle car il présente un modèle du modéle. Nous avons réalisé cette description en spécifiant les composants basés sur les patterns proposés par [**Hohpe**].

12.3 Domain Specific Langage

12.3.1 Présentation

L'utilisation d'un DSL implique, outre une définition et une implémentation des éléments du langage, la définition d'un *Controlleur* : le développeur de la solution va dans un premier temps décrire à l'aide du DSL les objets qu'il va utiliser pour implémenter sa solution. Le développeur décrit également les états initiaux et les propriétés de ces objets et les liens entre eux. Ensuite, sur base de cette configuration le *Controlleur* instancie les objets et active la plate-forme logicielle. Enfin, les objets s'activent sur base des données introduites c'est-à-dire les messages envoyés via les *Endpoint*s, dans cette plate-forme.

En nous ramenant à notre exemple, nous pouvons décrire un système de messagerie à l'aide de ce DSL. Puis nous devrions démarrer le *Control Bus* en lui fournissant cette description. Le *Control Bus* instanciera alors les différents objets et mettra en place le système qui sera alors prêt à recevoir les messages des applications à intégrer.

12.3.2 Types de DSL et mise en oeuvre

Selon [**FowlerDSL**], il existe deux types de DSL :
- **External DSL** : un langage défini dans un format qui n'est pas forcément celui du langage de l'implémentation des objets manipulés. Ce langage utilise par exemple un format XML alors que le langage d'implémentation est par exemple en Java ;
- **Internal DSL** ou **Embedded DSL** : le langage qui implémente le DSL est le même langage que celui utilisé pour l'implémentation des objets du domaine. La configuration de la solution est alors par exemple écrite en Java tout comme une implémentation des objets du domaine.

Quant à la mise en oeuvre, nous pouvons nous référer à l'article de [**vanDeursenKlintVisser**] qui en décrit les différentes phases que nous reprenons en citation.

> " **Analysis** (1) Identify the problem domain. (2) Gather all relevant knowledge in this domain. (3) Cluster this knowledge in a handful of semantic notions and operations on them. (4) Design a DSL that concisely describes applications in the domain.
>
> **Implementation** (5) Construct a library that implements the semantic notions. (6) Design and implement a compiler that translates DSL programs to a sequence of library calls. **Use** (7) Write DSL programs for all desired applications and compile them."

12.3.3 Implémentation

Nous devons dès lors choisir un type de format pour notre langage. Nous pouvons nous orienter vers un *Embedded DSL* mais cela implique que les utilisateurs de ce DSL connaissent également ce langage. De plus, cela lie le DSL avec l'implémentation des objets du domaine.

Nous pouvons aussi nous orienter vers un *External DSL* avec la possibilité de définir un format qui nous soit propre mais cela implique que nous écrivions un analyseur et un interpréteur de ce langage.

Enfin, nous pouvons opter pour un *External DSL* qui utilise un format de fichier relativement standard. Nous proposons d'utiliser à cette fin le format [**XML**]. L'utilisation de ce langage permet l'utilisation d'outils tels que analyseurs et parseurs relativement bien éprouvés. De plus, en utilisant un fichier *XSD*, nous pouvons définir une grammaire qui nous permette de valider la définition de la solution de façon statique et de détecter d'éventuelles erreurs avant le démarrage de notre système de messagerie.

12.4 Méta-modélisation

12.4.1 Introduction

Lors de la modélisation des objets, nous avons pu spécifier une partie de ceux-ci tout en laissant l'autre partie à spécifier en fonction des besoins. Nous avons par exemple mentionné l'interface *IApplicationInteface* mais nous n'avons spécifié avec précision les méthodes de la couche d'interface offerte par le *Endpoint* pour dialoguer avec l'application.

Pour éviter un développement classique entièrement confié aux développeurs, de plus en plus d'initiatives proposent de spécifier les développements en faisant appel au langage de modélisation UML. Nous nous penchons donc sur les standards actuels de modélisation.

12.4.2 Modèle en couche

Une classification des modèles a été définie par l'[**OMG**], un consortium international. Cette classification reprend les 4 niveaux suivants :

1. objets utilisateurs : objets du monde réel ;
2. objets du modèle : on parle de modèle. il s'agit des objets du monde applicatif c'est-à-dire les instances des objets créés dans une application ;
3. objets qui définissent les objets informatiques : on parle alors de metamodèle. Les classes d'objets font partie de ce niveau et enfin
4. les objets qui définissent les classes elles-mêmes. On parle alors de meta-metamodèle.

12.4.3 Outils et mise en oeuvre

Le concepteur des objets d'un domaine va spécifier son modèle d'objets au niveau du méta-modèle en se basant généralement sur le langage UML. Pour réaliser cela, il utilise des outils qui lui permettent de spécifier graphiquement les objets, leurs propriétés et leurs relations. Lors de cette étape, il peut également définir les contraintes d'intégrité et invariants qui caractérisent les objets. Lorsque la spécification est complète, l'utilisateur de l'outil peut générer une partie du code. Cette génération se fait maintenant sur base d'un fichier XMI, un fichier XML qui respecte un standard et qui reprend la description faite par le développeur. Le fichier XMI est utilisé comme base dans un ensemble de transformations qui aboutissent à la génération du code. Selon le niveau de complexité, le code est plus ou moins complètement généré. Il reste alors au développeur à finaliser ce code.

Pour réaliser ce travail, nous avons principalement utilisé l'outil StarUml (`http://www.staruml.com/`) qui propose les outils de description des objets en UML 2 et permet l'introduction de contraintes [**OCL**].

Nous avons également utilisé l'outil GME *Generic Modeling Environment* (`http://www.isis.vanderbilt.edu/projects/gme/`). Ce second outil, moins connu que le premier, permet par contre après modélisation des objets du domaine, de définir une implémentation d'un modèle du domaine. La vue graphique permet donc de bien illustrer la situation modélisée. Cette vue semble moins facilement possible en utilisant StarUml.

Il existe également un framework pour la plate-forme, Eclipse Graphical Modeling Framework (`http://www.eclipse.org/modeling/gmf/`), qui peut être mis en oeuvre avec le générateur de code Acceleo (`http://www.acceleo.org/pages/accueil/fr`) qui propose de générer du code en JEE et .net en transformant les modèles UML.

12.5 DSL

La génération du DSL au format XMLSchema est le résultat de transformations des objets décrits à l'aide de l'outil de modélisation. Ces objets, lorsque l'outil le permet sont exportés au format XMI. Le format XMI est une spécification de l'[**OMG**]. Elle a pour but une sérialisation des définitions au format UML (et OCL le cas échéant).

Pour réaliser la description de ce language, nous nous sommes basé sur le diagramme de classe de la figure 12.1 où les composants sont repris. Sur cette figure, le *System* est mis en évidence car il s'agit de l'élément "racine" de notre système. Il apparaît en fond noir et lettres blanches. Apparaissent également les composants qui sont instanciables. Ils ont un fond blanc. Enfin, les composants abstraits sont indiqués en fond gris.

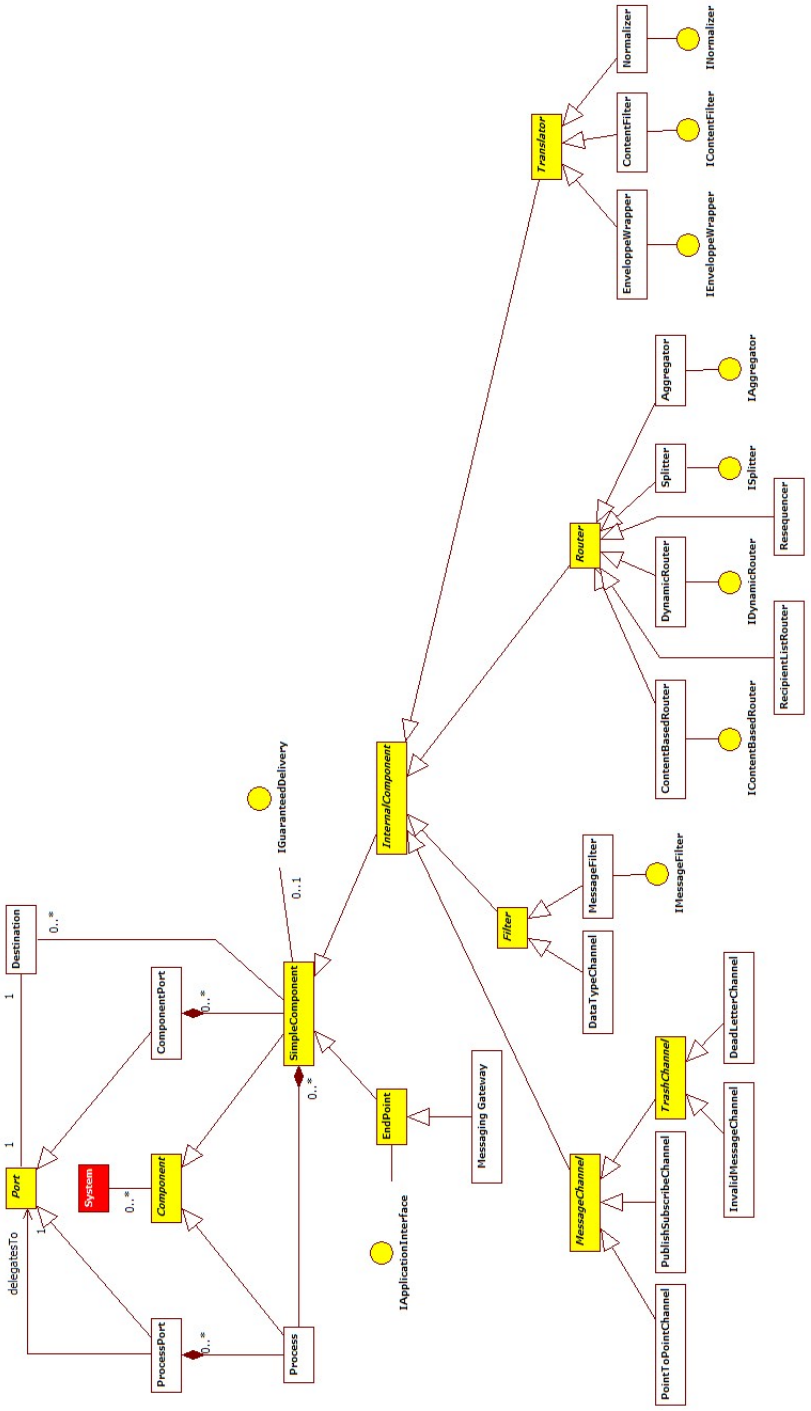

FIG. 12.1 – Diagramme de classes : Modèle

Nous construisons d'abord les types abstraits puis comme en programmation standard, nous créons les différents types qui les spécialisent. En y ajoutant les interfaces et les attributs, nous obtenons une structure très semblables à la structure hiérarchique de notre diagramme de classe.

A titre d'exemple, nous reprenons à la figure 12.2 la représentation graphique du *SimpleComponent*.

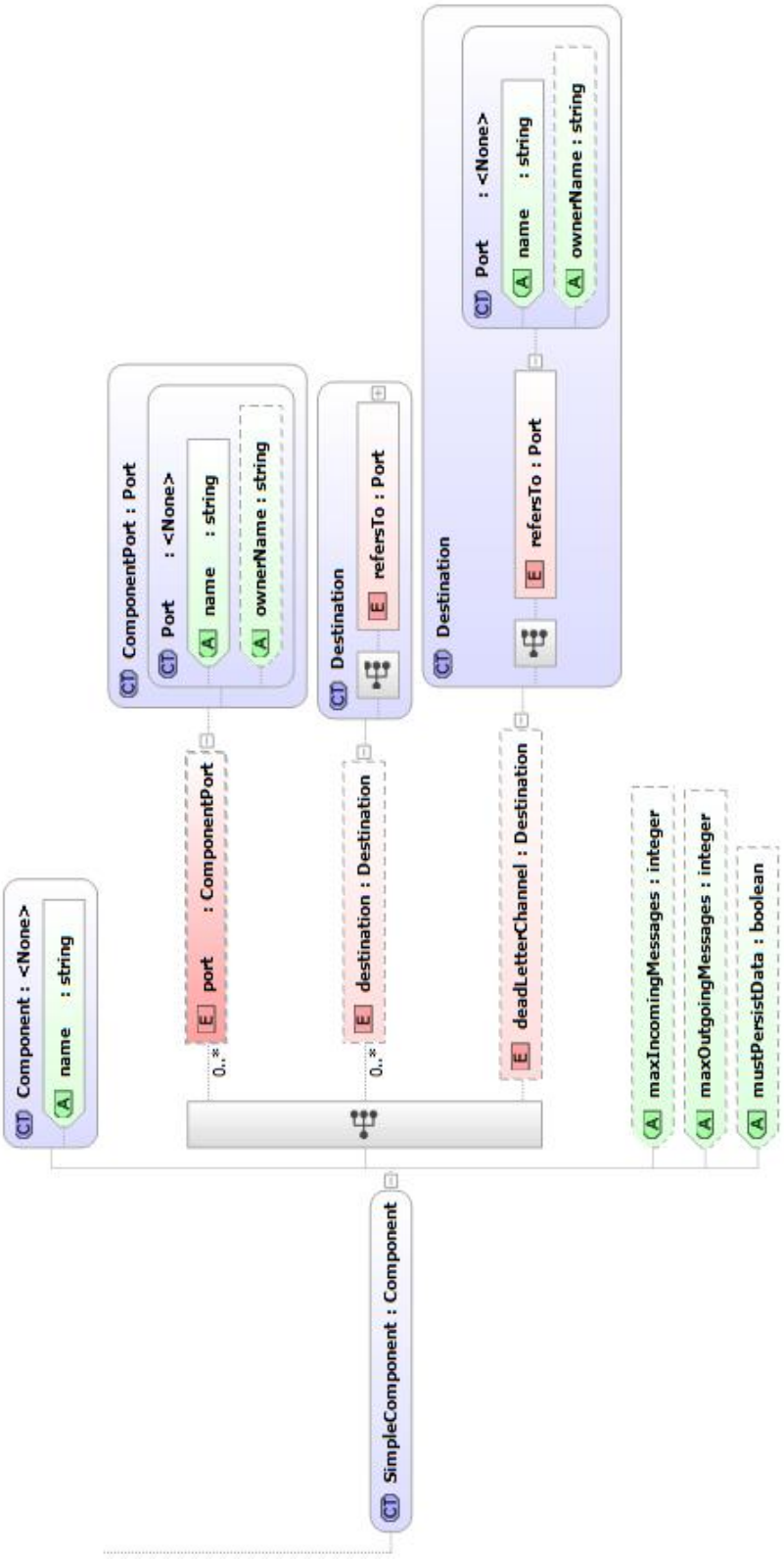

FIG. 12.2 – Représentation du SimpleComponent

Le fichier complet de la définition du DSL se trouve sur le CD-ROM qui accompagne ce travail. (voir en annexe le contenu de ce CD-ROM). Et sa représentation sous une forme textuelle :

```
<xs:complexType name="SimpleComponent" abstract="true">
    <xs:complexContent mixed="false">
      <xs:extension base="Component">
        <xs:sequence>
          <xs:element minOccurs="0" maxOccurs="unbounded"
             name="port" type="ComponentPort" />
          <xs:element minOccurs="0" maxOccurs="unbounded"
             name="destination" type="Destination" />
          <xs:element minOccurs="0" maxOccurs="1"
           name="deadLetterChannel" type="Destination" />
        </xs:sequence>
        <xs:attribute name="maxIncomingMessages"
         type="xs:integer" use="optional" />
        <xs:attribute name="maxOutgoingMessages"
         type="xs:integer" use="optional" />
        <xs:attribute name="mustPersistData"
         type="xs:boolean" use="optional" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

Chapitre 13

Discussion

13.1 Autres type de composants

La liste des composants que nous avons décrits n'est pas exhaustive. Nous pensons par exemple à un type de *Router* qui assurerait le routage des messages en fonction des ports sur lesquels il reçoit les messages.

Ce type de routeur pourrait être utilisé pour des processus comme la validation par applications successives de validateurs comme présenté dans le *Routing Slip* : les messages à valider entreraient par un port puis seraient expédiés vers un premier validateur qui renverrait son message-réponse sur le deuxième port du *Router* où tout message arrivant serait envoyé vers un deuxième validateur, ...

13.2 Améliorations du découplage

Nous avons vu dans ce travail que les composants n'utilisent en général qu'un seul port d'entrée. Nous pourrions définir pour ceux-ci un second port qui serait chargé de la gestion des messages de configuration. De même, au sein d'un *SimpleComponent*, nous utilisons un seul *MessageProcessor*. Pour des raisons de performance, il serait peut être utile de découpler le traitement des messages "business" des messages de configuration.

13.3 Amélioration de la réutilisation des processus

Nous n'avons pas vu d'outil ou de méthodes pour réutiliser un processus. Dans notre DSL, il faudrait prévoir un mécanisme d'importation de processus définis dans d'autres fichiers. Ainsi, nous pourrions utiliser un processus dans

plusieurs DSL.

13.4 Mise en place de contraintes dans la définition du langage

Nous avons choisi de définir notre langage en utilisant le format de fichier XMLSchema. Nous n'avons pas utilisé toute la puissance de ce langage. Il est notamment possible d'y placer des expressions XPath, expressions qui permettraient notamment de vérifier que tous les composants du système ont des noms uniques.

13.5 Amélioration de la recherche

Nous n'avons malheureusement pas pu approfondir nos connaissances à propos de *MOF*. Ce sujet serait intéressant à approfondir dans le cadre d'une méta-modélisation des composants et surtout au niveau de leur structure interne. Il en est de même pour le langage [**OCL**]. Enfin, de plus en plus d'articles font référence à la spécification [**QVT**] pour la transformation de modèle en langage générique. Ce sujet mériterait aussi quelques approfondissements pour le design et l'implémentation des composants.

13.6 Remarque personnelle à propos des *Endpoint*

Nous avons trouvé parfois une ambiguité au niveau de la description du *Endpoint*.

[**Hohpe**] présente au début de son livre le *Endpoint* comme permettant le dialogue asynchrone entre une application et le système et semble insister sur le découplage entre la requête et la réponse.

Par contre, lorsqu'il présente le *Service Activator* comme un *Endpoint*, il propose d'offrir un service synchrone qui bloque l'application appelante jusqu'à ce qu'elle reçoive la réponse à sa demande. Peut être faudrait-il dissocier le *Service Activator* des *Endpoint*s qui semblent fonctionner en mode asynchrone.

Ceci a été la source de problèmes de compréhension qui ont conduit aux définitions de mode de communication, de *Endpoint* client ou fournisseur, notions introduites dans la présentation des *Endpoint* et pas lors de leur analyse en profondeur. Le *Service Activator* avec ses possibilités d'appel synchrone et asynchrone a néanmois pu être modélisé sans trop de problème.

Pour une raison un peu semblable, nous avons créé une nouvelle catégorie de composants en regroupant les filtres *Datatype Channel* et *Message Filter* qui filtrent tous deux les messages mais se trouvaient dans des catégories différentes.

Chapitre 14

Conclusion

Nous avons dans ce travail, à partir de la description de patterns décrits dans le cadre d'une intégration d'applications dans une entreprise, modélisés les éléments qui peuvent servir de briques à la définition et la mise en place d'un middleware orienté messages. Cette définition nous a conduit à proposer un *Domain Specific Language* utile à la description d'un tel middleware.

La définition des composants est relativement simple et les noms proposés donnent une bonne idée de leur rôle. Grâce à la définition du DSL, il devrait être possible d'implémenter rapidement des solutions relativement légères d'intégration. Nous avons toutefois vu que des solutions déjà beaucoup plus abouties de middleware existent. Nous n'avons malheureusement pas eu assez de temps pour investiguer parmi celles-ci afin de voir si elles pouvaient être décrites à l'aide d'un DSL.

Chapitre 15

Annexes

15.1 Détails à propos de l'exemple introductif

Cette annexe reprend les acteurs de notre exemple, la description des messages échangés et un diagramme d'échange de messages sous la forme de diagramme de collaboration ; ce diagramme a été repris dans le texte du travail en utilisant les pictogrammes proposés par [**Hohpe**].

15.1.1 Acteurs et messages échangés

- Le propriétaire et/ou conducteur du véhicule en excès de vitesse ;
- Un « agent » chargé de collecter et transmettre les données de l'excès de vitesse au Tribunal ; cet agent sera appelé dans la suite de cet exposé la « Centrale des Infractions » pour évoquer sa fonction de collecte ;
- Le département des immatriculations (en abrégé D.I.V.) responsable de la gestion des immatriculations des véhicules automobiles ;
- Le Tribunal de Police compétent pour le jugement des excès de vitesse notamment ;
- La Poste qui est depuis peu chargée d'obtenir le paiement des amendes.

15.1.2 Messages échangés

- *ExcesVitesse* : Date et heure du constat, lieu, vitesse maximale autorisée, vitesse constatée, numéro d'immatriculation ;
- *DemandeIdentificationPropriétaire* : numéro d'immatriculation ;
- *IdentificationPropriétaire* : numéro d'immatriculation, nom du propriétaire, adresse du propriétaire ;
- *InformationsExcesVitesse* : *ExcesVitesse*, *IdentificationPropriétaire* ;
- *ProcesVerbal* : informationsExcesVitesse ;

- *PropositionTransaction* : *InformationExcesVitesse*, montantAmende ;
- *AccordTransaction* : *InformationExcesVitesse*, montantAmende ;
- *RefusTransaction* : *InformationExcesVitesse*, montantAmende ;
- *ConvocationProces* : *InformationExcesVitesse*, [refusTransaction], date du passage en jugement, lieu ou va se dérouler la séance ;
- *Jugement* : *InformationExcesVitesse*, date du passage en jugement, lieu de la séance, décision, montant amende
- *Amende* : informationExcesVitesse, Montant de l'amende.

15.1.3 Diagramme d'échanges de messages

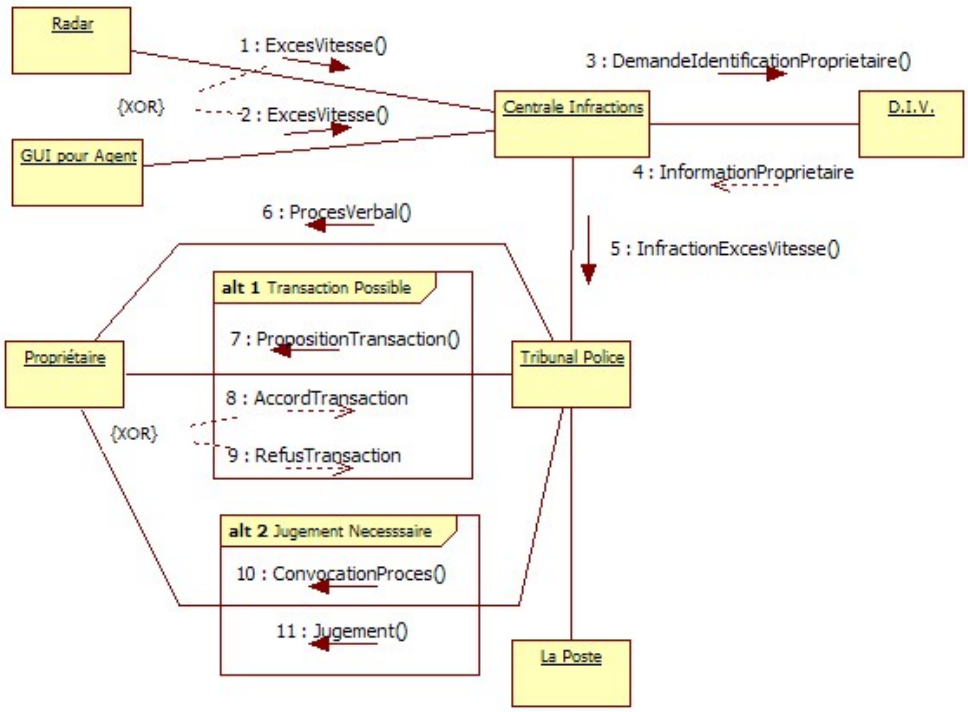

FIG. 15.1 – Diagramme de collaboration : échanges de messages entre applications

15.2 DSL : Exemple d'implémentation d'un MOM

Nous présentons un exemple élémentaire de MOM : Les informations arrivent dans un *Endpoint* client d'où ils sont envoyés vers un *PointToPoint-Channel* pour être envoyés vers un *Endpoint* fournisseur.

```
<?xml version="1.0" encoding="utf-8"?>
<system xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="D:\Memoire\Textes\DSL\Integration.xsd">

  <component xsi:type="MessagingGateway" name="Point d'entrée">
  <destination >
  <refersTo xsi:type="ComponentPort"
   name="Port du channel"
   ownerName="Un Channel"></refersTo>
      </destination>
  <applicationInterface implementation="1.app.interface"/>
    </component>


  <component xsi:type="PointToPointChannel" name="Un Channel">
    <port name="Port du channel"
     ownerName="Un Channel" />
    <destination>
       <refersTo xsi:type="ComponentPort"
         name="Port du point de sortie"
         ownerName="Point de sortie" />
     </destination>
    </component>

  <component xsi:type="MessagingGateway"
 name="Point de sortie">
    <port name="Port du point de sortie"/>
  <applicationInterface implementation="2.app.interface"/>
    </component>
</system>
```

15.3 Vues détaillée de l'architecture utilisée pour la création d'un DSL

Nous pouvons trouver ici les vues plus détaillées du schéma global des classes de notre modèle. Les vues sont rangées de bas en haut et de gauche à droite par rapport à la vue globale.

15.3.1 *System* et *Component*

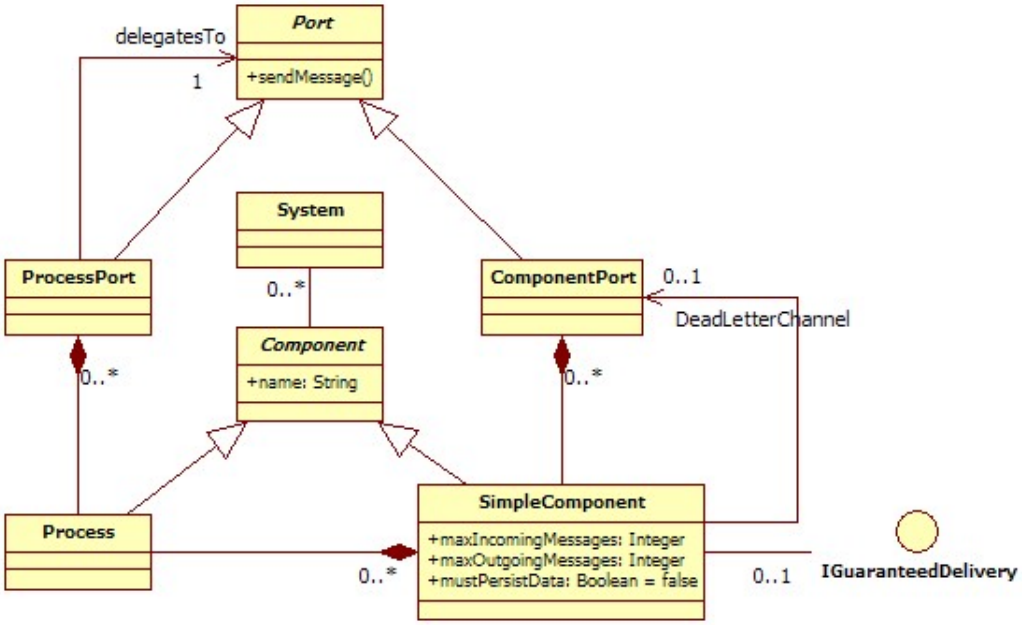

FIG. 15.2 – Diagramme de classes : *System* et *Component*

15.3.2 *Endpoint*

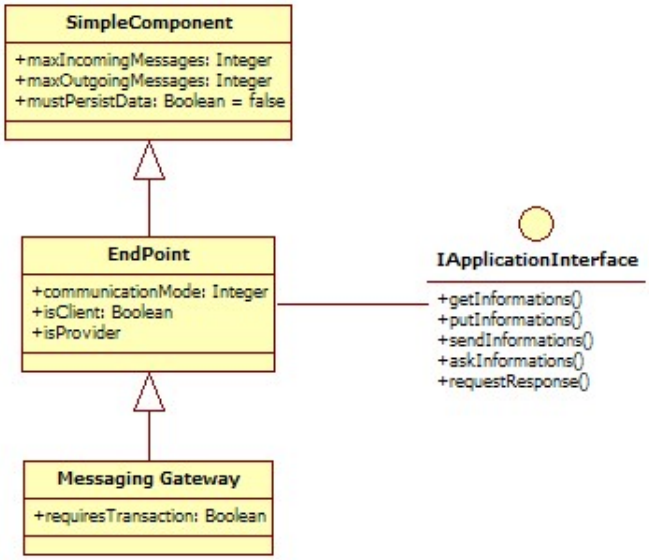

FIG. 15.3 – Diagramme de classes : *Endpoints*

15.3.3 *MessageChannel*

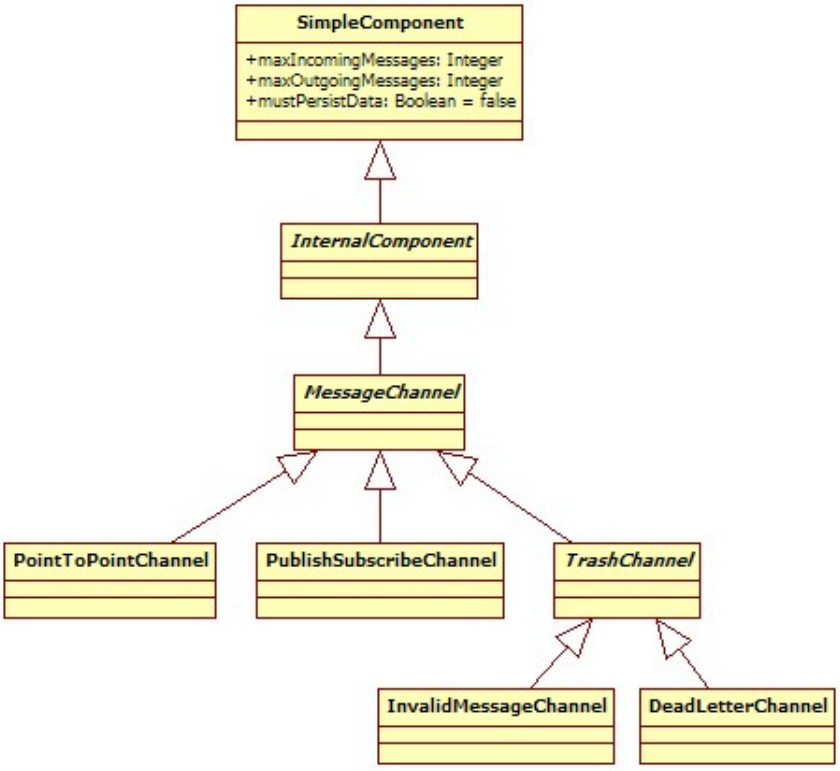

FIG. 15.4 – Diagramme de classes : MessageChannel

15.3.4 *Filter*

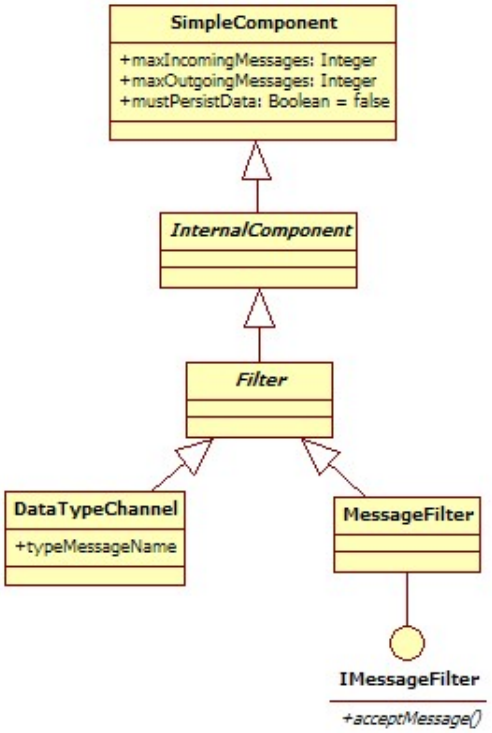

FIG. 15.5 – Diagramme de classes : *Filter*

15.3.5 *Router*

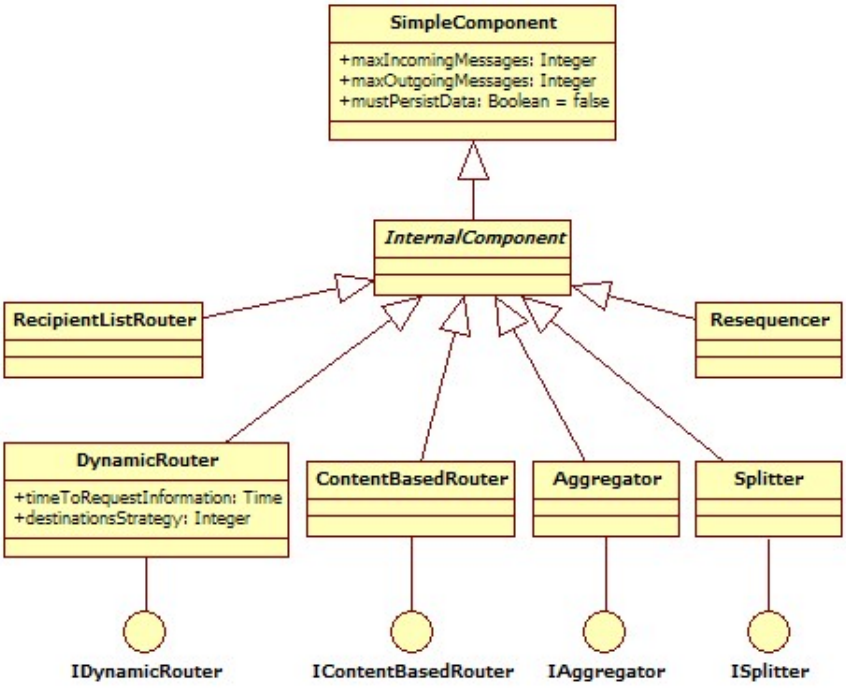

FIG. 15.6 – Diagramme de classes : *Router*

15.3.6 *Translator*

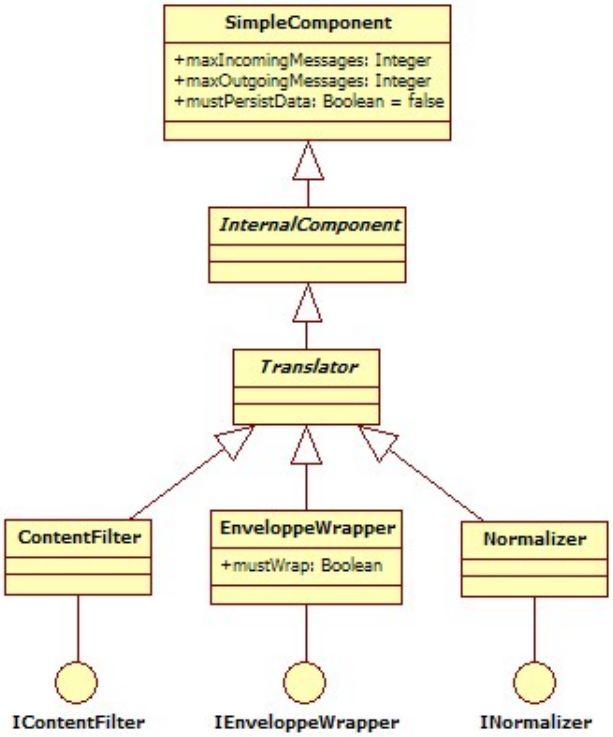

FIG. 15.7 – Diagramme de classes : Translator

15.4 Exemples de message de configuration utilisables par le *ControlBus*

configurationName	Description	Paramètres
`AddDestination`	Ajouter une destination à une liste de destinations	*Destination*
`RemoveDestination`	Retirer une destination de la liste des destinations	*IdDestination*
`UpdateDestination`	Modifier une destination de la liste	*IdDestination, Destination*
`UpdateTYMFilter`	Mettre à jour le type de message accepté par un *DataType Channel*	*Nom du type de message*
`SetInvalidChannel`	Fixer un *Invalid Message Channel*	*Destination*
`UnsetInvalidChannel`	Supprimer un *Invalid Message Channel*	n.a.
`SetMaxTime`	Fixer le temps maximum de conservation d'un message	*time*
`SetDestinationIDM`	Fixer la destination des messages envoyés vers un *Invalid Message* ou un *Dead Letter*	*Destination*
`UnsetDestinationIDM`	Supprimer la destination des messages envoyés vers un *Invalid Message* ou un *Dead Letter*	n.a.
`SetDeadLetterChannel`	Fixer le *Dead Letter Channel* pour un *Sender*	*Destination*
`UnsetDeadLetter-Channel`	Supprimer le *Dead Letter Channel* pour un *Sender*	n.a.

Contenu du CD-ROM

Vu la taille de certains fichiers, il a paru plus utile d'accompagner ce document d'un CD-ROM. Voici les fichiers contenus sur ce disque :

- FluxNotationHohpe.JPG : schéma qui reprend les échanges de message dans l'exemple introductif en utilisant les pictogramme proposés par [**Hohpe**] ;
- AllClasses.jpg : Diagramme de classes qui reprend toutes les classes utiles pour la définition du DSL ;
- Components.jpg : Diagramme de classes qui reprend les classes de base des composants ;
- MessageChannels.jpg : Diagramme de classes qui reprend les classes relatives aux *Channel*s ;
- Filters.jpg Diagramme de classes qui reprend les classes relatives aux *Filter*s ;
- Routers.jpg : Diagramme de classes qui reprend les classes relatives aux *Router*s ;
- Translators.jpg : Diagramme de classes qui reprend les classes relatives aux *Translator*s ;
- Integration.xsd : le fichier de définition du DSL ;
- SimpleMOM.xml : l'exemple également repris en annexe d'un simple système d'intégration.

Bibliographie

[**ChappellESB**] David A. Chappell,
Enterprise Service Bus,
O'Reilly,
United States of America,
2004,
ISBN 0-596-00675-6

[**vanDeursenKlintVisser**] Arie Deursen, Paul Klint, Joost Visser,
Domain-specific languages : an annotated bibliography,
ACM SIGPLAN Notices, 2000, volume 35, pages 26 à 36

[**EnglebertHeymans**] Vincent Englebert, Patrick Heymans, *Towards More Extensible MetaCASE Tools*,
University of Namur,
Computer Science Department,
PRECISE Research Center in Information Systems Engineering,
Rue Grandgagnage 21, B-5000 Namur, Belgium

[**FowlerLanguage**] Martin Fowler,
Language Workbenches :
The Killer-App for Domain Specific Languages ?,
`http://martinfowler.com/articles/languageWorkbench.html`

[**FowlerDSL**] Martin Fowler,
DomainSpecificLanguage,
`http://martinfowler.com/bliki/DomainSpecificLanguage.html`

[**GOF**] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides,
Design patterns, Catalogue de modèles de conception réutilisables,
Traduction de Jean-Marie Lasvergères,
Addison-Wesley, Vuibert, Paris, 1999, ISBN 2-7117-8644-7

[**Hohpe**] Gregor Hohpe, Bobby Woolf,
Enterprise Integration Patterns : designing, building, and deploying messaging solutions,
Addison-Wesley,
2004,
ISBN 0-321-20068-3

[**HTTP**] *HTTP - Hypertext Transfer Protocol*,
`http://www.w3.org/Protocols/`

[**Java**] `http://java.sun.com/`

[**MML**] José Álvarez, Universidad de Málaga, Spain,
Andy Evans, University of York, UK,
Paul Sammut, University of York, UK,
MML and the Metamodel Architecture,
`http://www.cs.york.ac.uk/puml/mmf/SammutWTUML.pdf`

[**MOF**] *OMG's MetaObject Facility*,
`http://www.omg.org/mof/`

[**OCL**] *Object Constraint Language, OMG Available Specification, Version 2.0, formal/06-05-01*,
`http://www.omg.org/docs/formal/06-05-01.pdf`

[**OMG**] *The Object Management Group (OMG)*,
`http://www.omg.org`

[**QVT**] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*,
July 2007,
`http://www.omg.org/docs/ptc/07-07-07.pdf`

[**SCA**] *Service Component Architecture Home*,
`http://www.osoa.org/display/Main/Service+Component+Architecture+Home`

[**Volter**] Markus Völter,
Architecture as Language : A story,
27 février 2008,
`http://www.infoq.com/articles/architecture-as-language-a-story`

[**W3C**] *World Wide Web Consortium*,
`http://www.w3.org/`

[**WSDL**] *Web Services Description Language (WSDL)*,
`(version1.1)http://www.w3.org/TR/wsdl`

[**XML**] *Extensible Markup Language*,
`http://www.w3.org/XML/`

[**XMLSchema**] *XML Schema*,
`http://www.w3.org/XML/Schema`

[**XQuery**] *An XML Query Language*,
`http://www.w3.org/TR/xquery/`

[**XSD**] *XML Schema*,
`http://www.w3.org/XML/Schema`